A Homogeneous Hierarchical Scripted Vector Classification Network with Optimisation by Genetic Algorithm

Hamish M. Wright B.E. (Hons)

A thesis submitted in partial fulfilment
of the requirements for the degree of
Master of Engineering
in
Electrical and Computer Engineering
at the
University of Canterbury,
Christchurch, New Zealand.

August 2007

ABSTRACT

A simulated learning hierarchical architecture for vector classification is presented. The hierarchy used homogeneous scripted classifiers, maintaining similarity tables, and self-organising maps for the input. The scripted classifiers produced output, and guided learning with permutable script instruction tables. A large space of parametrised script instructions was created, from which many different combinations could be implemented. The parameter space for the script instruction tables was tuned using a genetic algorithm with the goal of optimizing the networks ability to predict class labels for bit pattern inputs.

The classification system, known as Dura, was presented with various visual classification problems, such as: detecting overlapping lines, locating objects, or counting polygons. The network was trained with a random subset from the input space, and was then tested over a uniformly sampled subset.

The results showed that Dura could successfully classify these and other problems. The optimal scripts and parameters were analysed, allowing inferences about which scripted operations were important, and what roles they played in the learning classification system. Further investigations were undertaken to determine Dura's performance in the presence of noise, as well as the robustness of the solutions when faced with highly stochastic training sequences. It was also shown that robustness and noise tolerance in solutions could be improved through certain adjustments to the algorithm. These adjustments led to different solutions which could be compared to determine what changes were responsible for the increased robustness or noise immunity.

The behaviour of the genetic algorithm tuning the network was also analysed, leading to the development of a super solutions cache, as well as improvements in: convergence, fitness function, and simulation duration. The entire network was simulated using a program written in C++ using FLTK libraries for the graphical user interface.

ACKNOWLEDGMENTS

Firstly, I would like to thank my supervisor, Dr. Russell Y. Webb, for his constant guidance, patience, and contribution to this thesis; his immense capacity for generating and exploring ideas has been a source of inspiration. Also my co-supervisor Assoc. Prof. Philip J. Bones for his sound advice and for sharing with me his great knowledge of engineering.

Finally I would like to thank my parents and family for their continued love and support and to whom I owe so much.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

Chapter 1

INTRODUCTION

This thesis presents a novel learning network hierarchy for solving pattern classification problems. The goal of this thesis project was not only to implement a successful classification algorithm, but to determine important learning operations from an analysis of genetic algorithm selection results. Classification, which is the task of assigning objects to one of several predefined categories, is a pervasive problem that encompasses many diverse applications, including: identifying sea-bed objects based on SONAR recordings [1], classifying seismic signals [2], and speech recognition [3]. The goal of a learning classification algorithm is to both fit the input data and correctly predict the category labels of records it has never seen before. Therefore, a key objective of the learning algorithm is to build models with good generalisation capability [4]. Learning classification systems are an exciting and potentially extremely powerful field of engineering. They can provide solutions to problems that can not be reasonably solved manually.

Two major tools investigated and utilised in this thesis were genetic algorithms (GAs) and self-organising maps (SOMs). These provide some of the main learning capabilities of the network and are discussed in general, as well as their specific application to this project.

GAs, which were used extensively in this project, are an important method in learning systems. Many applications for GAs have already been identified, including: robotics [5], financial applications [6], pattern recognition [7], unmanned flight [8], and control and signal processing [9].

SOMs, which are effectively a density estimation tool, were also used extensively in this project. SOMs have been identified as having potential in many fields of engineering, including: forecasting electricity consumption [10], face recognition [11], visualising Windows viruses [12], monitoring of paper quality [13], and timber classification [14].

This thesis explores combining learning techniques such as GAs and SOMs, to implement a novel, hierarchical, homogeneous, scripted classifier network to solve visual classification problems. Visual image recognition has many applications in engineering, including: object recognition for automatic handling, security, palm recognition, and fingerprint recognition [15].

1.1 THE NETWORK

A network, hereafter referred to as Dura, is introduced in this thesis. Initial development of the network architecture and implementation in C++ was carried out by Dr. Russell Webb. It was then developed and completed by myself. The final design, simulation data, and analysis of the network presented is my own work.

Dura is a learning classification network that requires training on a labelled input space. After sufficient training the network would attempt to predict the classifications for previously unseen inputs. Dura employs SOMs and similarity tables to organise the inputs and outputs between classifiers. Scripted instructions control the learning and output of the classifiers for every input. The scripted instructions were permuted using a GA to find the optimal instruction set to correctly classify the input space.

The network is hierarchical; it consists of several layers of classifiers. Each layer can only interact with the layers directly above and below. Only the bottom layer sees the actual input pattern. Only the top layer sees the actual label for the current input (and only during the training period).

The network is homogeneous; every classifier in the network has exactly the same script tables. During training, however, each classifier could develop its own SOM and similarity table in order to communicate with the layers above and below. An in depth overview of the Dura network is given in chapter 3.

The concept for the hierarchical nature of the Dura network stemmed from the theory that a human mind may build a description of an object in a hierarchical manner, abstracting information at each step. Initially, coarse classifications may be made, such as big or small, or round or sharp. Then, gradually more specific classifications may be made, such as has wheels, or legs. Finally a classification is arrived at, such as car, or person. Dura implements a representation of this hierarchical classification with information compression at each level, attempting to automatically abstract the important information as it is passed through the network.

One of the key concepts is that the network script tables are highly flexible. The solutions found by the GA could completely bypass the SOM or the similarity table, or employ a number of different implementations with a large space of behaviour defining parameters.

1.2 MOTIVATION AND STATEMENT OF PURPOSE

The purpose of this project was to create a novel classification architecture hierarchy and to analyse its performance, and potential, in vector classification. Dura uses a novel scripted instruction architecture. A large space of highly parametrised instructions was created, *cf.* Section 3.4.1. The combinations of these instructions implemented by the classifiers in the network were then tuned using a GA. This method allowed a very

large instruction space to be investigated and tested, generating extremely complex relationships are been unlikely to be manually encoded. The GA allowed empirical analysis of the resulting instruction space "solutions", revealing which instructions were important, and which were not, for a particular problem space. Therefore, one of the main motivations for this thesis is to show that creating an adaptable learning instruction architecture, and tuning it with a GA, is a potentially powerful technique in learning systems.

Several important questions emerged: which scripted actions were important for learning in the hierarchy and why? What different modes of learning emerged and what roles did they play? What sort of problems could the network solve? How adaptable was the network to different input problem spaces? Did the architecture offer any computational gains over other classification algorithms? Throughout the research these and other questions were addressed and investigated.

1.3 OUTLINE OF THESIS

In Chapter 2 a background is given on the primary learning mechanisms which were GAs, and SOMs. In Chapter 3 an overview of the Dura network is given, explaining the key features, principles, and flow of information in the network. Chapter 4 presents the experimental methods undertaken, including the problem spaces investigated and an overview of the tools used to run the network simulations. A short summary and overview of the C++ code used to create the program that simulates the network is also given. Simulation results for each of the problem spaces are presented and analysed in Chapter 5. Attention is given to discussing the resulting script tables of the best solutions and what could be learned from them. Certain observations during simulations which led to developments and improvements in the algorithm are also discussed. Attention is given to the robustness and performance in the presence of noise for the solutions found. Also discussed in Chapter 5 are investigations into the behaviour of the GA itself. These investigations led to several interesting developments: the super-solutions cache (SSC), an improved fitness function, and reduced simulation time. Chapter 6 gives a brief summary of the important findings of this thesis, while expanding on these ideas and discussing their significance and implications. Suggestions future development of the project is also provided.

Chapter 2

BACKGROUND

This chapter provides background information and references for two of the main learning features in Dura which were GAs, and SOMs. Particular attention is given to GAs which were an extremely important tool. For a detailed overview of machine learning and pattern classification several excellent resources are suggested: *Introduction to Machine Learning* [16], *Pattern Recognition and Machine Learning* [17], and *Machine Learning: An Artificial Intelligence Approach* volumes I-IV [18, 19, 20, 21].

2.1 GENETIC ALGORITHMS

2.1.1 History

In the 1960s, three different interpretations of evolutionary computing were developed in three different places leading to several significant publications: *Artificial Intelligence through Simulated Evolution* [22] by L.J. Fogel, *Evolution Strategies* [23] by I. Rechenberg, and *Adaption in Natural and Artificial Systems* [24] by J.H. Holland. J.H. Holland's work is of particular interest for introducing the genetic algorithm. Since then, the principles discussed have been developed and applied to many different fields of engineering.

Another interesting branch of evolutionary computing is genetic programming, which was introduced in the 1980s and popularised by J. Koza [25] in the 1990s. In genetic programming, a computer program is evolved to optimise a solution to a given problem. Dura behaves in a similar way, optimising scripted instruction tables to produce the best network to classify a problem space. It is important to note that GAs were not used during the training of an individual network, the goal of the GA was to find the optimal network from a population of Dura network individuals. A Dura network individual is an entire classification network with scripted instruction tables as described in Chapter 3.

Because GAs are a fundamental tool in the Dura project, a general introduction to GAs is given in this chapter.

2.1.2 Overview

GAs are essentially an optimisation tool emulating Darwinian survival of the fittest evolution dynamics [26]. Potential solutions to a given problem are referred to as individuals of a population. Each individual is made up of a set of values, or genes, that define that individual's characteristics. In this thesis, the GA individuals are Dura networks, the genes are the script instruction tables as discussed in Chapter 3. How well an individual solves the optimisation problem is the individual's fitness. This fitness is important, because it determines the individual's likelihood of survival and mating. It is important to note that GAs do not seek to replicate Darwinian evolution in its entirety (for example, in a GA an individual may live forever). There are many different implementations of the GA, the following is an outline of a typical scheme.

The algorithm begins with an initial population of individuals or solutions, these are usually randomly generated. The fitness of each is evaluated using the fitness function. Next, the population is usually ranked in order of fitness. The crucial step of the GA occurs when individuals are selected, based on their fitness, for crossover. When two individuals have been selected, they become parents and crossover their genes to form new children solutions. In this way the "better" genes of the fittest individuals are more likely to be passed on and to be tested in different combinations with other "good" genes. New individuals can also be created by the mutation process. In mutation, a small number of an individual's genes will be altered in a random fashion. This has the effect of introducing new potentially useful genes into the gene pool.

The new individuals are then passed into the next generation of the GA. At this point, a number of the fittest individuals from the previous generation are also passed into the next generation; this process is known as elitism. Usually, a number of new randomly generated individuals will also be introduced into the next generation. This has a similar effect to mutation, by injecting new genetic material, while also maintaining the population size. Finally, the next generation is evaluated using the fitness function and the entire process repeats, either for a certain number of generations or until a desired fitness is reached.

In each generation of the GA, through the crossover and mutation mechanisms, good genes, and good gene combinations are typically more likely to occur than in the previous generation. Therefore, over time the best individuals will be better solutions to the problem; hence, optimisation occurs. Implementation details are given in the following sections for the main GA operations: fitness function, ranking, crossover, and mutation.

2.2 FITNESS FUNCTION

The fitness function is the operator which determines how well an individual solves the given problem. The fitness function is specific to a given problem and must therefore be implemented individually for each GA. An example is optimisation of a sports results predictor, where the fitness of an individual is determined by how many results of a sample set were predicted correctly.

In Dura, the fitness was determined by the number of correct classifications divided by the total number of classifications made, as shown by Equation 2.1.

$$Fitness = \frac{NumCorrectClassifications}{TotalNumClassifications} \tag{2.1}$$

An improved fitness function was devised to penalise solutions which only predicted a label which was overrepresented in the input space; this effect is discussed in Section 5.1.2.

2.3 RANKING

Ranking is achieved by first ordering the population by fitness, then assigning a new fitness based on the individual's rank. Ranking has several useful properties which can combat potential pitfalls in GAs. It ensures that small differences between fitness levels are expanded, which helps prevent stagnation. Ranking also attenuates very large differences between fitness levels, which helps prevent premature convergence [27].

After ranking, the fitness distribution is usually determined by a selection pressure, β. A low selection pressure results in an even distribution, with all individuals having the same probability of mating. A high selection pressure results in only the fitter individuals getting a high probability of mating. Ranking can be either linear or non-linear.

In the equations that follow, β is the selection pressure, N is the population size, ($N \geq 2$), i is the i^{th} individual after ranking ($i = 0$ being the least fit, $i = N - 1$ being the fittest individual).

2.3.1 Linear Ranking

In linear ranking, the fitness value for an individual is calculated by Equation 2.2.

$$Fitness(i) = 2 - \beta + \frac{2(\beta - 1)i}{N - 1} \tag{2.2}$$

where the selection pressure is in the range 1.0-2.0. Figure 2.1 illustrates the fitness assignment for a linear ranking with various selection pressures and a population size of 10.

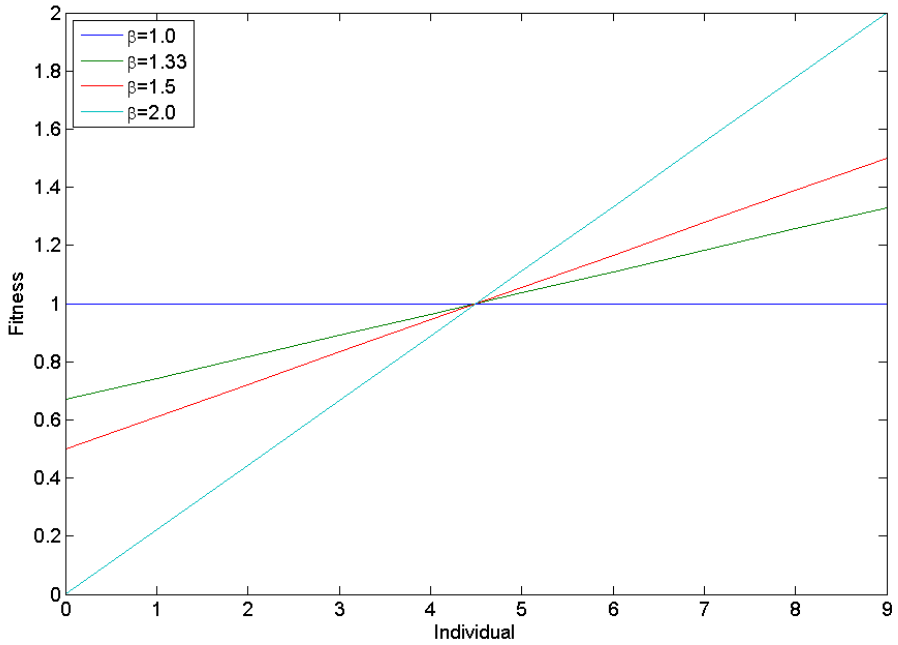


Figure 2.1 Fitness assignment for linear ranking with different selection pressures, β.

2.3.2 Non-Linear Ranking

In non-linear ranking, an individual's fitness is determined by a non-linear operator acting on its rank. In Dura the fitness of each individual is found by Equation 2.3.

$$Fitness(i) = \frac{1}{\beta^{(N-i-1)}} \tag{2.3}$$

where $\beta > 0$.

Figure 2.2 illustrates the fitness assignment for non-linear ranking with various selection pressures and a population size of 10.

Non-linear ranking provides greater control over the fitness assignment and higher selection pressures. This results in the best individuals receiving a much greater relative fitness than with linear ranking.

2.4 BOLTZMANN SELECTION

In the Boltzmann selection scheme, individuals are assigned a new fitness based on Equation 2.4, with an adjustable selection pressure, β.

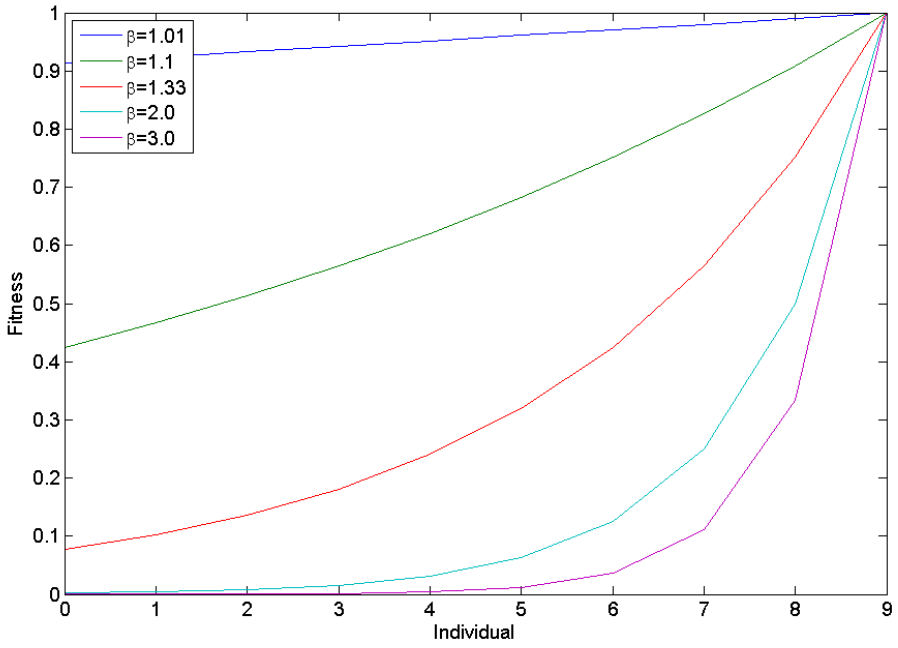


Figure 2.2 Fitness assignment for non-linear ranking with different selection pressures, β.

$$Fitness' = e^{\beta Fitness} \tag{2.4}$$

where $\beta > 0$.

In Boltzmann selection, the selection pressure is slowly increased over time, with the goal of allowing the GA to progressively focus its search [28].

2.5 SELECTION

Selection is the process by which individuals are chosen to be parents for crossover. The probability of an individual being selected for crossover is usually found by normalising its final fitness by the total fitness of the population. Selection is a very important step in the GA process and a wide variety of schemes exist. These include, roulette wheel, stochastic universal sampling, local, truncation and tournament selection. An excellent, mathematically derived, comparison of selection algorithms is described in *A comparison of selection schemes used in genetic algorithms* [29] by T. Blickle and L. Thiele.

2.5.1 Roulette Wheel Selection

Roulette wheel is one of the most basic and common selection schemes. It is also known as stochastic sampling with replacement. In roulette wheel selection the individuals occupy contiguous segments of a line; the size of an individual's segment is its chance to crossover. Next, a random number between 0 and 1 is generated. The individual's segment which falls under the random number is selected. This repeats until two unique individuals are selected. Consider a population that has been linearly ranked with $\beta = 2.0$, as shown in Table 2.1.

Table 2.1 A population ranked using linear ranking and a selection pressure of 2.0.

Individual	Fitness	Mating Probability
0	0.000	0.000
1	0.222	0.022
2	0.444	0.044
3	0.667	0.067
4	0.889	0.089
5	1.111	0.111
6	1.333	0.133
7	1.556	0.156
8	1.778	0.178
9	2.000	0.200

Figure 2.3 illustrates the roulette wheel for this population.

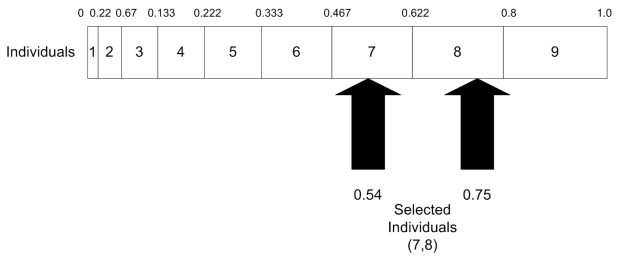


Figure 2.3 Roulette wheel for linearly ranked population.

Two random numbers were generated, 0.54 and 0.75, to select the parents for crossover. The roulette wheel selection algorithm provides a zero bias but does not guarantee minimum spread. Empirical evidence suggests that sampling bias can have a significant impact on GA performance [30].

2.5.2 Stochastic Universal Sampling

Stochastic universal sampling is similar to roulette wheel selection, but is modified to provide zero bias and minimum spread. The individuals are first mapped to a contiguous line in the same fashion as roulette wheel selection, as illustrated in Figure 2.4.

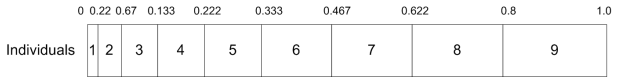


Figure 2.4 Each individual's fitness is mapped on to a contiguous line.

Now, instead of using random numbers to select the individuals, two evenly spaced locations along the line are found. A random number, r, between 0 and 0.5 is generated indicating the first location. The second individual is found by moving a distance of exactly 0.5 from the first individual. Consider the linearly ranked population from Table 2.1, where $r = 0.325$. Figure 2.5 illustrates the selection process.

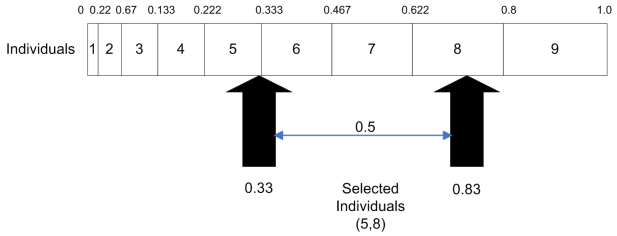


Figure 2.5 Stochastic universal sampling selection example.

The process is repeated until the required number of crossovers have occurred. Alternatively, a pool of N parents can be found by generating r, where $0 \leq r \geq \frac{1}{N}$, using this as the location for the first parent and then moving a distance $\frac{1}{N}$, $N - 1$ times.

Consider the following example, using the same population shown in Figure 2.5, where $N = 4$ and $r = 0.15$. Parents are selected from the following locations: 0.15, 0.40, 0.65, and 0.90 (individuals 4, 6, 8, and 9). For certain problems such as the travelling salesman problem, stochastic universal sampling is shown to be the selection method of choice. Because of zero bias and minimum spread, stochastic universal sampling ensures a selection of individuals which is closer to what is deserved than roulette wheel selection [31].

2.5.3 Local Selection

In local selection the population is mapped in some way, usually on to a 2-dimensional toroidal grid. Once an individual has been selected as a parent, the second parent can only be chosen from a region of neighbouring individuals. Offspring produced then replace one or both parents. An example population is illustrated in Figure 2.6 showing the region of selection for a primary parent.

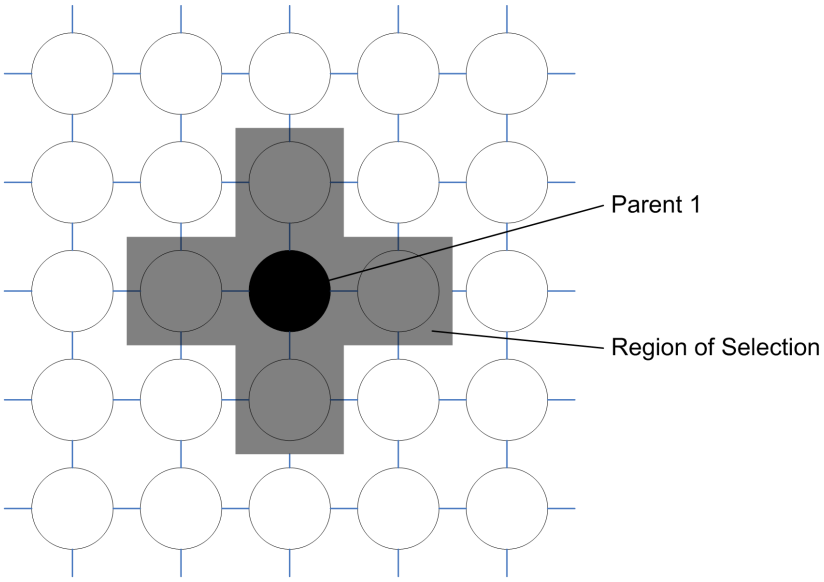


Figure 2.6 Local selection example. The potential region of selection for the central individual is shaded grey.

Every individual's neighbourhood overlaps with other nearby neighbourhoods. This overlap of neighbourhoods provides an implicit mechanism for genetic material to migrate through the population. There are several approaches to local selection, the effect of changing the size and shape of neighbourhoods is not well understood and is examined empirically in [32].

2.5.4 Truncation Selection

In truncation selection only the fittest individuals are contenders for crossover. After the population has been ranked, a truncation threshold is applied to the population. Individuals above the threshold are selected at random for mating as illustrated in Figure 2.7.

One of the drawbacks with truncation selection is that it leads to a much higher loss of diversity for the same selection intensity when compared to ranking and tournament selection [33].

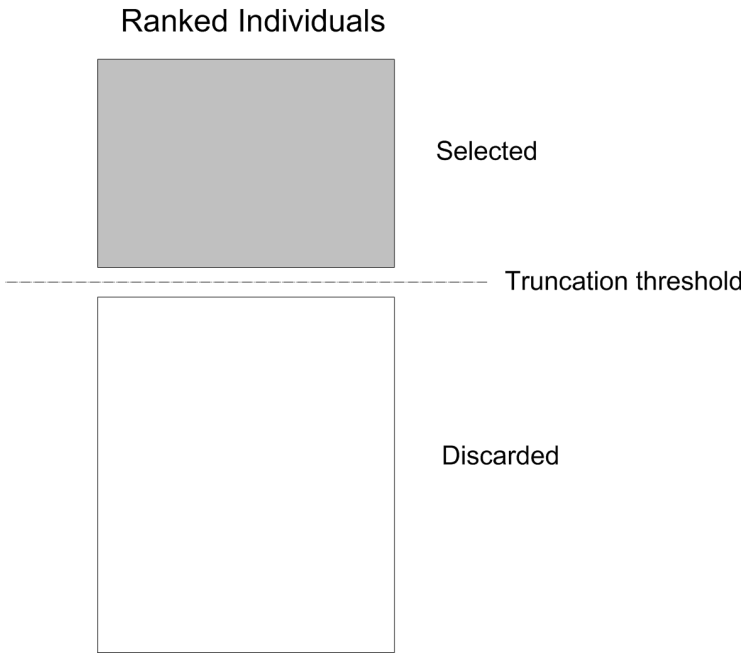


Figure 2.7 Truncation selection example. The population is ranked, then any individuals above the threshold are selected and paired at random.

2.5.5 Tournament Selection

In tournament selection a number of individuals are taken from the population at random. The number of individuals is known as the tour size. The individual with the greatest fitness in the tour subset is selected as a parent. This process is then repeated as many times as parents are required. Consider a population size of nine with a tour size of three as illustrated in Figure 2.8.

In tournament selection, the larger the tour size the greater the selection pressure. Tournament selection can also lead to sampling bias, an unbiased tournament selection scheme is discussed in [30].

2.6 CROSSOVER

Crossover is the operation of combining two parent individuals' characteristics to create new children individuals, as illustrated in Figure 2.9. Crossover can be referred to by several terms, including: hybridisation, recombination and mating. The key principle of crossover is that the new individuals may be better solutions than either of the parents; that is, they may exhibit a better combination of the parents' chromosomes.

The occurrence of crossovers in a GA is usually controlled by the crossover rate, ranging from 0 to 1.0. The crossover rate usually determines the percentage of genes to crossover between individuals. For some implementations, a crossover rate of 0.5

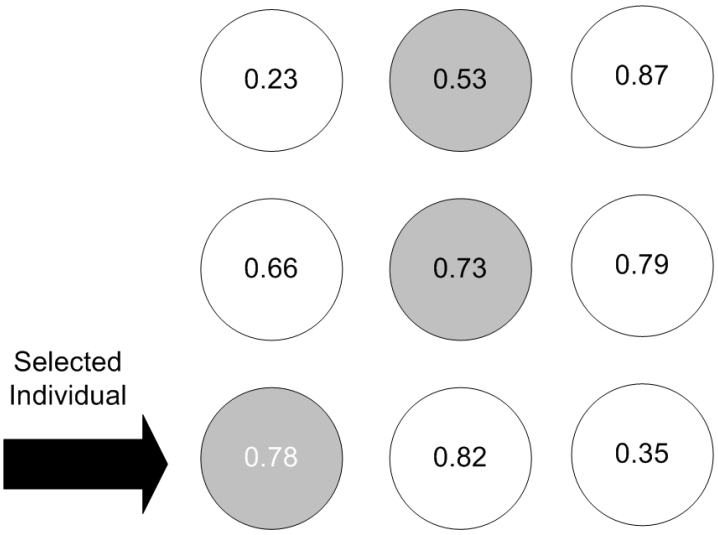


Figure 2.8 Tournament selection example. The greyed individuals have been randomly selected to be in the tour sub-set. The individual with the greatest fitness in the tour sub-set is selected as a parent.

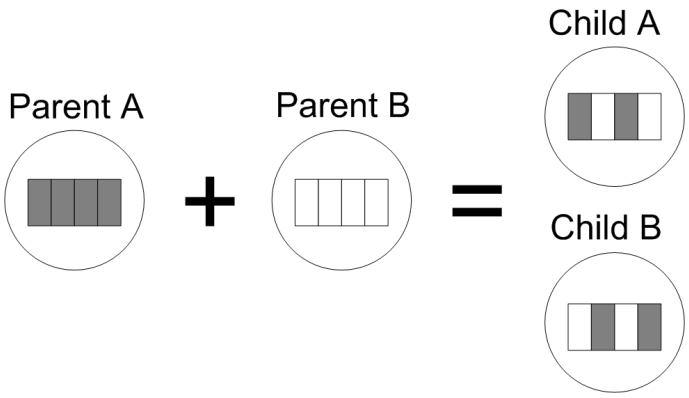


Figure 2.9 Selected parents crossover their genes to create new offspring.

provides the maximum crossover effect.

There are several different implementations for crossover depending on the problem and the specific representations of an individual's genes, including: one point, two point, uniform, arithmetic, and heuristic.

2.6.1 One Point

In one point crossover, two parent individuals are chosen. A single crossover point is then randomly selected along the chromosome, as illustrated in Figure 2.10.

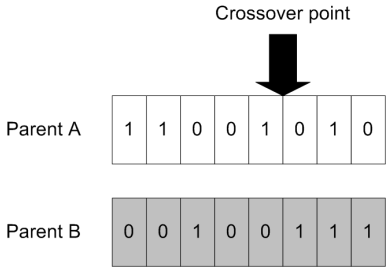


Figure 2.10 Parent chromosomes with single crossover point indicated.

The children's chromosomes are produced by swapping the genes from the beginning of the parents' chromosomes up to the crossover point, as illustrated in Figure 2.11.

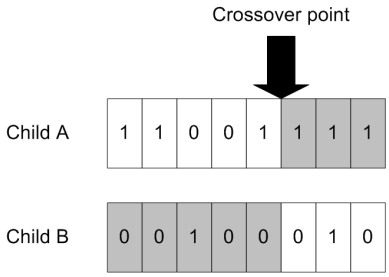


Figure 2.11 Resulting chromosomes after one point crossover.

2.6.2 Two Point

In two point crossover, two parent individuals are chosen and two crossover points are then randomly selected along the chromosome as illustrated in Figure 2.12.

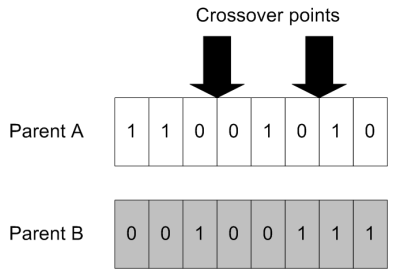


Figure 2.12 Parent chromosomes with single crossover point indicated.

Figure 2.13 shows how the children's chromosomes are produced by swapping the parents' genes between the crossover points.

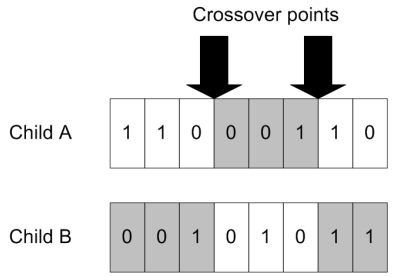


Figure 2.13 Resulting chromosomes after two point crossover.

In both one point and two point, entire sequences of genes can be preserved, allowing the propagation of complex characteristics. The number of crossover points used can be any arbitrary number greater than one. However, one or two point crossover is generally sufficient for most problems.

2.6.3 Uniform

In uniform crossover, individual genes are swapped according to some probability, usually referred to as the mixing ratio. Consider the original parents' chromosomes, as illustrated in Figure 2.14.

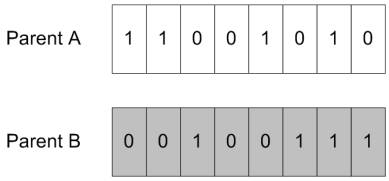


Figure 2.14 Parent chromosomes before crossover.

If a mixing ratio of 0.5 is used then the offspring will receive an approximately even contribution of genes from each parent, as illustrated in Figure 2.15.

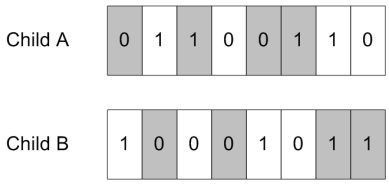


Figure 2.15 Resulting chromosomes after uniform crossover.

Uniform crossover allows the parent chromosomes to be mixed at the gene level rather than the segment level (as with one and two point crossover). Useful long sequences of genes can be disrupted or lost, however, for some problems, the flexibility of uniform crossover outweighs the disadvantages.

2.6.4 Arithmetic

In arithmetic crossover, offspring are produced by linear combinations of the parents. Two arithmetic crossover equations are given in Equation 2.5 and Equation 2.6.

$$Offspring_1 = a \cdot Parent_1 + (1 - a)Parent_2 \tag{2.5}$$

$$Offspring_2 = (1 - a)Parent_1 + a \cdot Parent_2 \tag{2.6}$$

where a is a random weighting factor (chosen before each crossover operation from 0 to 1). Arithmetic crossover may be preferred when genes have continuous values. Consider the chromosome illustrated in Figure 2.16.

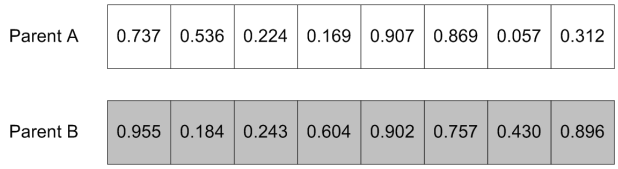


Figure 2.16 Parent chromosomes before crossover.

If $a = 0.613$, then the offspring illustrated in Figure 2.17 would be produced.

Figure 2.17 Resulting chromosomes after arithmetic crossover.

2.6.5 Heuristic

In heuristic crossover, the best and worst individuals in the population are selected as parents. The offspring are created according to Equation 2.7 and Equation 2.8.

$$Offspring_1 = Parent_{best} + r(Parent_{best} - Parent_{worst}) \tag{2.7}$$

$$Offspring_2 = Parent_{best} \tag{2.8}$$

where r is a random number between 0 and 1. Consider the following example where Figure 2.18 shows the best and worst individuals in a ranked population.

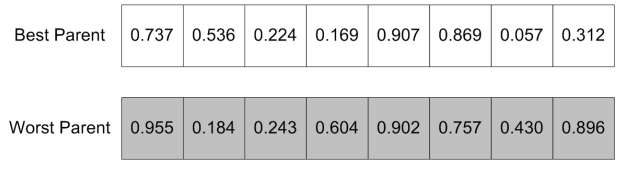


Figure 2.18 Best and worst parent chromosomes before crossover.

Next, r is randomly selected to be 0.110 producing the offspring shown in Figure 2.19.

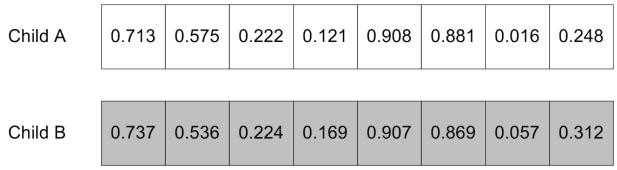


Figure 2.19 Offspring from a heuristic crossover.

Offspring1 may be regenerated if an invalid individual is created; this is possible due to upper and lower bounds. After a maximum number of retries, the offspring defaults to an exact copy of the worst parent.

2.7 MUTATION

Mutation is the second major mechanism by which solutions evolve in a GA. In mutation, a small percentage of the population have some of their characteristics altered to new random values. Mutation serves the purpose of maintaining diversity in the population, adding new potentially useful values into the gene pool and also preventing premature convergence. It is possible to achieve optimisation with a random walk through the search space using only the mutation operator.

The occurrence of mutation is controlled by the mutation rate, and must be carefully chosen. If it is too small premature convergence to a local optimum may occur. If it is too high it may lead to the loss of good solutions. Another option is to adjust the mutation rate dynamically while the algorithm is searching. As with crossover there are a number of different implementations for mutation, including swap, and random walk.

2.7.1 Swap Mutation

In swap mutation, existing genes are interchanged in an individual. There are three main forms of swap mutation: regular, adjacent, and sequence.

2.7.2 Regular Swap Mutation

In regular swap mutation, values to be swapped are selected at random across the chromosome, as illustrated in Figure 2.20.

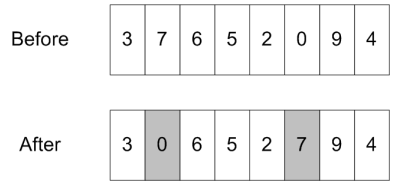


Figure 2.20 Example of regular swap mutation.

2.7.3 Adjacent Swap Mutation

In adjacent swap mutation, values selected to be swapped must be adjacent, as illustrated in Figure 2.21.

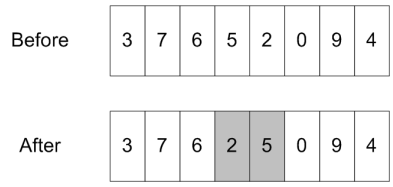


Figure 2.21 Example of adjacent swap mutation.

This technique may be appropriate when the genes are ordered along the chromosome in some meaningful way, or where adjacent genes are likely to be highly related.

2.7.4 Sequence Swap Mutation

In sequence swap mutation, sequences of a given length are randomly selected and swapped, as illustrated in Figure 2.22.

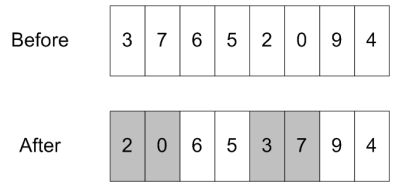


Figure 2.22 Example of sequence swap mutation.

This form of mutation preserves sequences of genes which may be important for certain problems.

2.7.5 Random Walk

In random walk a random number of genes are altered by some random value, as illustrated in Figure 2.23.

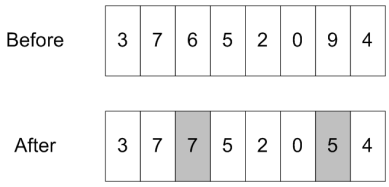
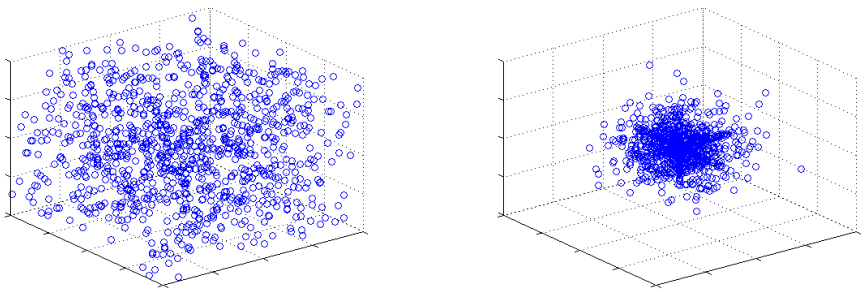


Figure 2.23 Example of random walk mutation.

2.8 DIVERSITY

Diversity refers to the average distance between individuals in a population. Usually euclidean distance is used, but other measures may also be appropriate. A population has high diversity if the average distance is large, a low diversity if the average distance is small. In Figure 2.24, the population on the left has high diversity, while the population on the right has low diversity.



(a) (b)

Figure 2.24 Gene distribution for two different GA populations, (a) has a high diversity and a large proportion of the input space is being searched, (b) has a low diversity and only a small proportion of the input space is being searched.

Diversity is essential to the GA because it enables the algorithm to search a larger region of the space. A relatively new simple algorithm for preserving diversity is described in [34]. After some time the GA search will converge and diversity in the population will tend to decrease. Too rapid, or too slow, convergence can be avoided

by careful choice of fitness function, selection method, and parameters such as mutation rate, and crossover rate.

2.9 GENETIC ALGORITHM DISCUSSION

For certain problems, GAs may have a tendency to converge toward local optima rather than the global optimum. GAs will usually find a relatively good solution for a complex search space quickly, but not necessarily the best solution. Opinion is divided over the importance of crossover versus mutation [35, 36]. It can be argued that crossover is nearly always equivalent to a large mutation and therefore redundant. Optimisation can be achieved using just one or the other. However, for a more robust and general algorithm both crossover and mutation are implemented. There are several alternative optimisation schemes to GAs, including: hill climbing [37], simulated annealing [38], and particle swarm optimisation [39]. In order for a GA to be effective there must be a way to automatically and accurately quantify the fitness of the solutions. For certain tasks it can be extremely difficult to implement a fitness function (for example, creating a musical composition). The implementation and evaluation of the fitness function is a major factor in the overall speed and efficiency of the algorithm. The choice of parameters such as mutation rate, crossover rate, and selection pressure can greatly effect the ability of the GA to converge [40].

2.10 SELF-ORGANISING MAPS

The SOM [41] is a form of artificial neural network (ANN) used for data clustering [42]. The network adapts to associate the output nodes with groups or patterns from an input data set. It is trained using competitive learning to produce a low dimensional representation of the training samples. A SOM compresses information while preserving the most important topological and metric relationships of the primary data elements in the display [43].

2.10.1 Structure

A network of n neurons is created, each with a weight vector of the same dimensionality, d, as the input vector. Each neuron in the network is fully connected to the input layer as shown in Figure 2.25. The neurons in the network can be arranged in many configurations, the most common being a 1 or 2-dimensional lattice. The specific structure of the SOMs used in this thesis is discussed in Section 3.1.3.

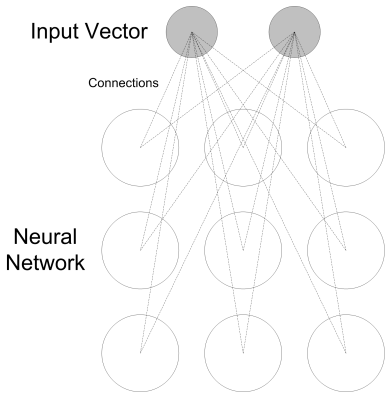


Figure 2.25 Structure of a small 2-dimensional SOM showing fully connected input where $n = 9$ and $d = 2$.

2.10.2 Operation

Firstly, the neurons' weight vectors, $\mathbf{W}$, have their elements initialised to random values. A training sample input, $\mathbf{X}$, is presented to the network and its distance to each of the weight vectors is calculated. Many different distance functions can be used here, such as euclidean distance, as given in Equation 2.9, or by the sum-of-products (SOP), as per Equation 2.10. A third distance function was also investigated in 5.1.1.

$$Distance = \sqrt{\sum_{i=1}^{n} (W_i - X_i)} \tag{2.9}$$

$$Match = \sum_{i=1}^{n} (W_i \cdot X_i) \tag{2.10}$$

The neuron which is closest to the current input becomes the SOM winner. The weights in the winner and the neurons in the neighbourhood of the winner are adjusted to be more like the current input according to Equation 2.11.

$$\mathbf{W}(t+1) = \mathbf{W}(t) + N(v,t)L(t)(\mathbf{X}(t) - \mathbf{W}(t)) \tag{2.11}$$

where $L(t)$ is the learning coefficient,
$N(v,t)$ is the neighbourhood coefficient,
v is the lattice distance to the SOM winner,
t is the iteration number.

The neighbourhood coefficient affects how much the weights are altered as a function of the neuron's lattice distance from the SOM winner. Early in the simulation the neighbourhood is broad and the self-organising takes place on the global scale. As the neighbourhood coefficient declines, the neurons' weights converge to local estimates.

The learning coefficient affects the magnitude of the adjustment of the weight vectors. Both the learning coefficient and neighbourhood coefficient decrease monotonically over time allowing the map to settle. For statistical accuracy, the number of iteration steps should be at least 500 times the number of network units [44]. Therefore, for an 8 node SOM, as in Dura, around 4000 iterations would be required. However this was not a strict guideline as the network used a novel SOM implementation. The default learning coefficient used in Dura is described by Equation 2.12. The default neighbourhood function used in Dura is illustrated in Figure 2.26.

$$L(t) = 1 - 0.0001t \tag{2.12}$$

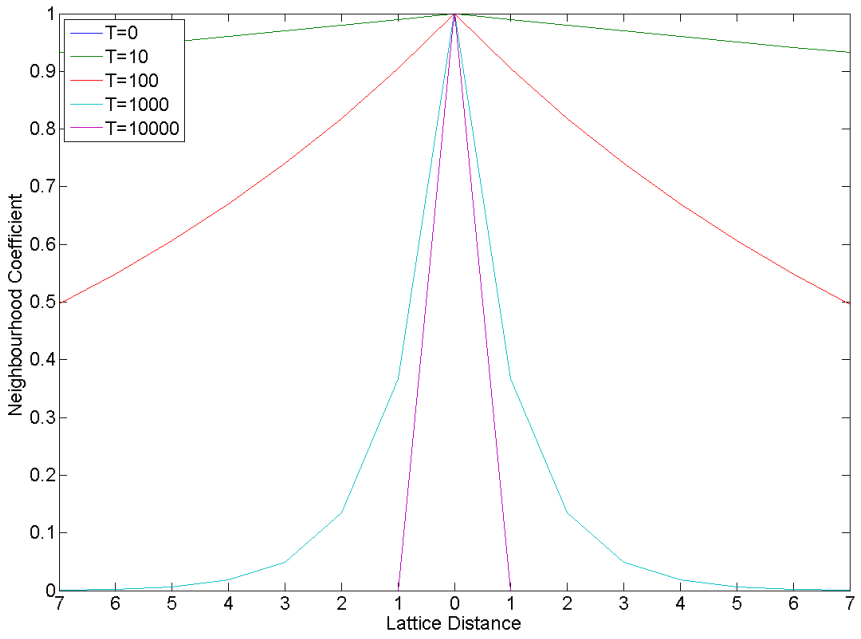


Figure 2.26 Dura SOM neighbourhood coefficient, $N(v,t) = 0.999^{vt}$.

Dura SOMs use eight neuron weight vectors with a 32-bit input pattern, giving a compression of 4 to 1. Consider Figure 2.27 which illustrates the SOM weights after training, an input pattern and the winning SOM node.

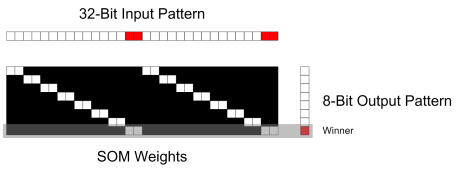


Figure 2.27 SOM example, showing the input retina, SOM node weights, and the winning node.

Figure 2.27 illustrates how a 32-bit input vector is compared to the SOM weights and which node is deemed to be the winner.

Chapter 3

DURA NETWORK

In this chapter a detailed background on the Dura network is given. The major components making up the network are discussed and details of implementation are given. This chapter also addresses the sequence of events that occur during a time step in a Dura network as well as in a generation of the GA. Running the GA for a number of generations to optimise a Dura network solution is referred to as a simulation. As stated in Chapter 1 initial development of the network architecture and implementation in C++ was carried out by Dr. Russell Webb. It was then developed and completed by myself. The final design, simulation data, and analysis of the network presented is my own work.

3.1 CLASSIFIER NODES

Scripted classifier nodes are the basic unit of the Dura network. A hierarchy of classifier nodes is constructed where informational binary signals are passed between nodes. At the bottom level the *input* signal is the input bit-pattern, at the top level the *output* signal is the classification of the current input bit-pattern. The actions performed by the classifier nodes at each time step are controlled by tables of script instructions. The script tables are homogeneous throughout the network. A common technique in classification is to divide the problem into sub-problems and assign a set of different function approximators or "experts" to each sub-problem [45, 46]. In the Dura network, each classifier node is identical. Unlike some hierarchical mixture of expert systems, the Dura network requires no a priori knowledge of the input space. Each classifier in the network has four script instruction tables: state, learning, request, and output. Each classifier considers four informational signals and also maintains a SOM and a similarity table as learning mechanisms. The SOM is used to perform density estimation on the *input* signal. This divides the input space into subsets which produce the same winning SOM node. The similarity table is used to keep a record of which *training* signals were given most frequently for the SOM winners. Figure 3.1 illustrates the internal components and signals of a scripted classifier node. The roles of signals,

script instruction table, the SOM, and the similarity table are discussed in the following subsections.

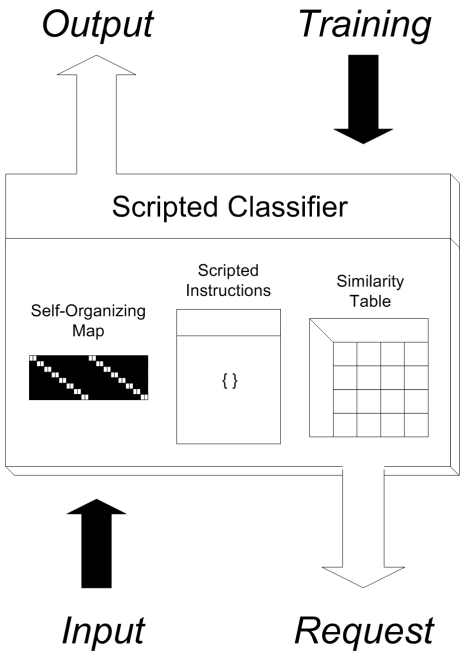


Figure 3.1 Classifier node showing internal components.

3.1.1 Signals

Each classifier has four signals associated with it: *input*, *training*, *output*, and *request*.

- *Input* is a 32-bit vector taken from the level's input pattern, referred to as the classifier's input retina.
- *Training* is the signal feeding back from the level above, implemented as an 8-bit vector. At the top level, the *training* signal is the correct classification label, the upper *request*, for the current input pattern.
- *Output* is the signal which makes up the input of the classifiers in the level above. Each classifier node produces an 8-bit vector for the *output*. At the top level the *output* is the predicted classification label for the current input pattern.
- *Request* is used to train the classifiers in the level below, or to indicate which *output* is expected. The *request* therefore becomes the *training* signal to classifiers below. Each classifier node can request from up to 4 other nodes. Therefore, the *request* is implemented as a 32-bit vector.

Typically the *training* and *output* signals will only have a single bit on. Each individual bit in the signal vectors is referred to as a category. For each classifier node, the *input* and *training* signals are fixed, the *output* and *request* signals are controlled using a table of script instructions.

3.1.2 Script Instruction Table

The script instruction table contains all the instructions to control a classifier node's behaviour and actions at each time step. The instructions are referred to as script instructions due to the nature of their implementation. The motivation behind script instruction tables was to allow different modes of behaviour to develop in the network and also to provide an architecture which could be easily tuned by a GA. As an example, a particular mode might allow the network to learn new inputs quickly, using a high learning rate. Then after some time the network might switch to another mode and reduce its learning rate to avoid the effect of noise. Later the network might begin to output incorrectly due to previously unseen inputs, at which point it can switch back into the first behavioural mode. The types of script instructions used in Dura are discussed in Section 3.3.

State scripts are used to index script instruction tables which are tuned by a GA to find the best scripts to solve a particular classification problem. In each classifier node there are `kNumStateBits` state scripts, and $2^{kNumStateBits}$ states. For each state, there is an associated script sequence for learning, outputting, and requesting. Each state script instruction returns either true or false and sets (to one or zero respectively) the corresponding state bit.

For each time step in the network the state of each of the classifier nodes is first determined. The state scripts are then executed in sequence and the state of the classifier is the value of its state bits. The state is then used as an index to execute scripts from the other script tables, as illustrated in Figure 3.2. Consider the following example conditions where there are two state scripts, `StateIsTopLevel` and `StateWinnerIsElementOfTraining`.

`kNumStateBits` = 2
`StateIsTopLevel` = true
`StateWinnerIsElementOfTraining` = true
state = $11_2 = 3_{10}$
Therefore the script instructions from row 3 would be used, as illustrated in Figure 3.2.

For each row in a script table (except the state script table) there may be one or more scripts to execute. The actual script instructions implemented in Dura are discussed in more detail in Section 3.3.

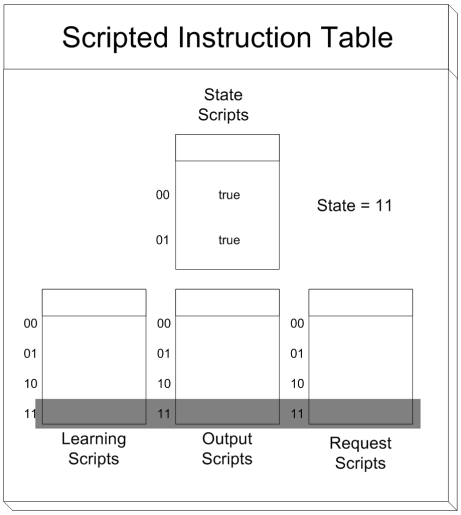


Figure 3.2 Script table operation, showing the state of the classifier node and the resulting indexed instructions selected from the script tables: learning, request, and output.

3.1.2.1 Script Instruction Tunable Parameters

The basic unit of the script instruction table is the `ScriptToken`. A `ScriptToken` consists of a unique `code`, and four optional tunable parameters. The `code` is used to look up the instructions function and the tunable parameters can be used to control some part of the instruction, such as a learning rate, or a threshold. The ability for scripted instruction parameters to be easily tuned by a GA is a key feature of the Dura project. The C++ definition for the `ScriptToken` struct is given in the following:

```
typedef struct {
    u32 code;
    s32 sarg1;
    s32 sarg2;
    float farg1;
    float farg2;
} ScriptToken;
```

An example `ScriptToken`, using the enumerated value `AugmentTrainingInWinner` for the `code`, is given in the following:

```
{ kAugmentTrainingInWinner, 0, 4, 0, 0.870 }
```

3.1.2.2 Bounding Tunable Parameters

A mechanism is implemented to bound the parameter values for each instruction. For certain instructions only some ranges of parameters are allowed (for example, a learn-

ing coefficient must be between 0.0 and 1.0). An `InstructionDefinition` contains the bounding information about each script instruction. The C++ definition for the `InstructionDefinition` struct is given in the following:

```
typedef struct {
    Instruction f;
    const char *description;
    S32Description s1des;
    S32Description s2des;
    FloatDescription f1des;
    FloatDescription f2des;
} InstructionDefinition;
```

The value `f` is a function pointer, indicating where in the program the instruction is actually implemented, `description` is a unique string used to identify the instruction. The `S32Description` and `FloatDescription` structs each store three values: the `upperBound`, the `lowerBound` for the respective parameter, and a boolean value to specify if the parameter is actually used in this instruction.

During the crossover process, the scenario of parameters going out of the specified range can occur frequently. When this happens the parameter is simply regenerated at random between the bounding values. This is equivalent to large mutations, which are generally more likely to be harmful than helpful to the individuals fitness. An example entry from the instruction definition table for a SOM learning operation is given in the following:

```
{ GenericClassifier::LearnBestAndNeighbors, ‘‘kLearnBestAndNeighbors’’,
{false, 0, 0}, {false, 0, 0}, {true, 0.0, 1.0}, {true, 0.0, 1.0} }
```

The above entry will be indexed in the instruction definition table at the value specified by the enumeration `kLearnBestAndNeighbours`. This particular instruction uses the function pointer `GenericClassifier::LearnBestAndNeighbours()` and two float parameters, ranging from 0.0 to 1.0. The two float parameters are used to control the neighbourhood coefficient and the learning coefficient for updating the SOM.

3.1.3 Self-Organising Map

Density estimation is performed on the *input* to each classifier node in the hierarchy by a 1-dimensional SOM. The SOM divides the *input* space into subsets which produce the same SOM winner. General SOMs are discussed in more detail in Section 2.10. The motivation for using SOMs is that they provide a mechanism to abstract the *input* information, extract features, and compress it as it is passed through the network.

SOMs can discover the topological relations and other abstract structures in the *input* signals [47]. The SOM performs basic classification on the input data by associating spatially similar *inputs* with the same SOM output. These initial SOM classifications are then used by the Dura node to make more complex decisions on the *output* to the level above.

The SOM implementation in Dura uses eight 32-bit weight vectors, generating a compression of 4 to 1. The SOM can operate in either a supervised or unsupervised mode. In the unsupervised mode the SOM acts as a bootstrapping mechanism and is used to provide the basis for decisions by the classifier. Figure 3.3 illustrates an *input* being processed by the SOM.

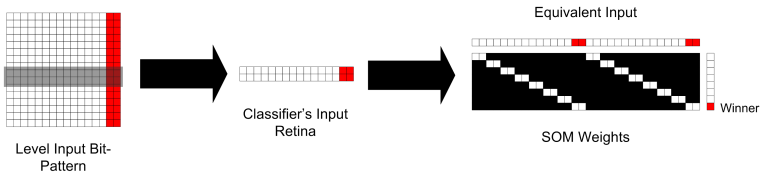


Figure 3.3 SOM configuration, showing the input bit-pattern, the region of selection, and the winning node.

The example SOM weights shown in Figure 3.3 are actual weights from a simulation test problem space containing only 8 possible inputs, *cf.* Section 4.2.2; white indicates a high weighting, black a low weighting. Each of the nodes in the SOM has adjusted to match one of the inputs. Therefore, the SOM has learned to generate a different winner for each of the eight inputs and has divided the input space in the most efficient way.

In addition to finding the SOM winner, the *input's* match to every SOM node is stored. The classifier's scripts can then use the relative strengths of the matches to determine its state and to make decisions for updating its learning, and *output* and *request* signals.

3.1.4 Similarity Table

Every classifier node in the network maintains a similarity table. The similarity table implements the modal of semi-supervised learning. The motivation for similarity tables is that each classifier node maintains a table of relational strengths between the signals it processes and uses those strengths to provide a basis for decision making.

In the basic mode each classifier node keeps a record of which *training* category was given for each SOM winner. In this way if a particular *training* category is consistently given for a particular SOM winner then the classifier can choose to output the most frequently trained category, rather than the SOM winner itself. This has the effect of

translating the SOM winner vocabulary into the vocabulary of the classifier above. At the top level this allows the *output* to be translated into the correct classification. The idea was then taken further to allow a broader range of signal and SOM relationships to be considered. The similarity table implementation in Dura is discussed in the following:

The Dura network uses 8-bit *output*, *request*, *training*, and SOM winner patterns. Each bit in the 8-bit patterns represents a unique category. Essentially, the similarity table maintains a list of relational strengths between the categories. For example: if the SOM winner is category `A`, find the relational strength of the *training* signal also being `A`. In an 8-bit architecture each classifier node maintains an 8-by-8 similarity table.

The similarity table is updated by specifying a row and column address to be incremented (augmentation). The row and column address can potentially be any permutation of the 8-bit signals used by the classifier. The Dura script instruction tables determine what the rows and columns represent for each Dura implementation. In the simplest case, the columns represent the SOM winner and the rows represent the *training* category. Consider Table 3.1 which illustrates a 4-bit architecture similarity table. If the SOM winner is category `A` and the *training* is category `B`, increment row `B`, column `A`.

Table 3.1 Similarity table augmentation example.

	A	**B**	**C**	**D**
A	0	0	0	0
B	***1***	0	0	0
C	0	0	0	0
D	0	0	0	0

The table can either be read as a direct count or normalised by the sum of the row or column. If the columns represent the SOM winner and the rows represent the *training* signal then the similarity table can be used to determine which category is being requested most frequently for a subset of *inputs*. This is done by simply finding the maximum value in the column of the winner category; this is called fingding the column similarity. Classifiers can perform several operations on the similarity table, such as finding the row similarity, or finding the column similarity of a `target` category.

3.1.4.1 Finding Column Similarity

Given a `target` category, a classifier node may require to find the category which has the highest similarity in that column. Consider the similarity table shown in Table 3.2 and Table 3.3. `A` is the `target` category. Using a direct count, `D` has the highest column

similarity, as shown in Table 3.2. Proportionally, `B` has the highest column similarity, as shown in Table 3.3.

Table 3.2 Finding the column similarity using a direct count example.

	A	**B**	**C**	**D**
A	6	7	12	0
B	5	1	1	1
C	0	3	15	0
D	***10***	21	11	1

Table 3.3 Finding the column similarity using relational proportions example.

	A	**B**	**C**	**D**
A	0.24	0.28	0.48	0.00
B	***0.63***	0.13	0.13	0.13
C	0.00	0.17	0.83	0.00
D	0.23	0.49	0.26	0.02

A classifier node can use either approach to determine the column similarity. The direct counts can also be used to determine the significance of the relationships. For example: if fewer than 100 entries in the table have occurred then the result can be ignored.

3.1.4.2 Finding Row Similarity

Row similarity can be found in precisely the same way as column similarity, replacing columns for rows. For clarity, another example is given. Consider the similarity table shown in Tables 3.4 and 3.5. `A` is the `target` category. Using a direct count `C` has the highest row similarity, as shown in Table 3.4. Proportionally, category `C` also has the highest row similarity, as shown in Table 3.5.

Table 3.4 Finding the row similarity using a direct count example.

	A	**B**	**C**	**D**
A	6	7	***12***	0
B	5	1	1	1
C	0	3	15	0
D	10	21	11	1

For the case where columns represent the SOM winner and rows represent the *training* signal, row similarity can be described as the SOM node which won most frequently for the given training signal.

Table 3.5 Finding the row similarity using relational proportions example.

	A	**B**	**C**	**D**
A	0.29	0.22	***0.31***	0.00
B	0.24	0.03	0.03	0.50
C	0.00	0.09	0.38	0.00
D	0.48	0.66	0.28	0.50

In the Dura network, the similarity table is not limited to update using the SOM winner and the *training* signal as indices. Any combination of SOM winner, *training*, column similarity of winner, column similarity of *training*, row similarity of winner, or row similarity of *training* is allowed. Furthermore, not only the winner but any of the SOM categories can be used depending on their relative match strengths to the current input. A large space of complex signal relationships are available for the Dura network to explore implemented with script instructions and tuned using the GA. This allows the Dura classifier network to determine which relationships are important without manual intervention and effort. Relationships that emerged frequently could then be analysed to determine their importance in the classification operation.

3.1.5 Internal Parameters

Classifier nodes also maintain several internal parameters which can be used to adjust their learning and output. Internal parameters are another way in which classifiers can control which mode of behaviour that they are in. Two internal parameters, confusion and plasticity, are discussed in the following subsections:

3.1.5.1 Confusion

One of the ways in which different modes of operation can emerge in the Dura network is through the confusion parameter. Each classifier maintains a confusion value between 0 and 1. The confusion can be increased if there is a mismatch between the current *training* signal and the current *output* signal. Then, a different mode of learning can be pursued when there is repeated disagreement between network levels. After a certain length of time, the confusion will fall below some threshold and a new learning or no learning mode can be entered. Confusion was maintained by the `UpdateConfusion` script described in Section 3.3

3.1.5.2 Plasticity

Plasticity is used to adjust the learning rate of the classifier nodes. Generally, over time the plasticity will decrease allowing the learning to dampen down. However, similarly

to the confusion parameter, after a sequence of incorrect classifications or disagreement between levels, it can be useful to increase the plasticity to promote new learning paths. Plasticty was maintained by the `UpdatePlasticity` script described in Section 3.3.

3.2 NETWORK CONFIGURATION

The network architecture is extremely flexible. It allows control of the flow of information between levels in the network and between individual classifier nodes. Each classifier node receives a 32-bit *input*, constructed from four 8-bit segments selected from the level input bit-pattern. At the bottom level, the level input pattern is the primary input pattern. At subsequent levels the level input pattern is constructed from the outputs of the classifiers in the level below. Each classifier node can request from up to four classifier nodes from the level below. Figure 3.4 illustrates an example two level network configuration with five classifier nodes.

The network connections are specified in the `gNetworkConfig` array at compilation time. A brief overview of the network configuration implementation is given here, the exact implementation details are not of significant importance. The `gNetworkConfig` array is simply an integer array with special syntax. The C++ array required to produce the network configuration shown in Figure 3.4. `#define` commands are used to make the code more readable and also more maintainable.

```
#define NONE 0xFFff
#define ALL_FROM(anchor) anchor, anchor+1, anchor+2, anchor+3
#define NO_FB() NONE, NONE, NONE, NONE
#define FB_TO(anchor) anchor, anchor+1, anchor+2, anchor+3

u32 gNetworkConfig[] = {
    4,
       ALL_FROM(0), NO_FB(), // 0
       ALL_FROM(2), NO_FB(), // 1
       ALL_FROM(4), NO_FB(), // 2
       ALL_FROM(6), NO_FB(), // 3
    1,
       ALL_FROM(0), FB_TO(0), // 4
    0
};
```

The first integer in the array indicates how many classifier nodes are to be in the first level of the network. For each classifier node, there is a set of four values indicating the indices from the level's input bit-pattern to use as its *input* signal, as illustrated in Figure 3.5. Another set of four values indicates which classifier nodes to request from.

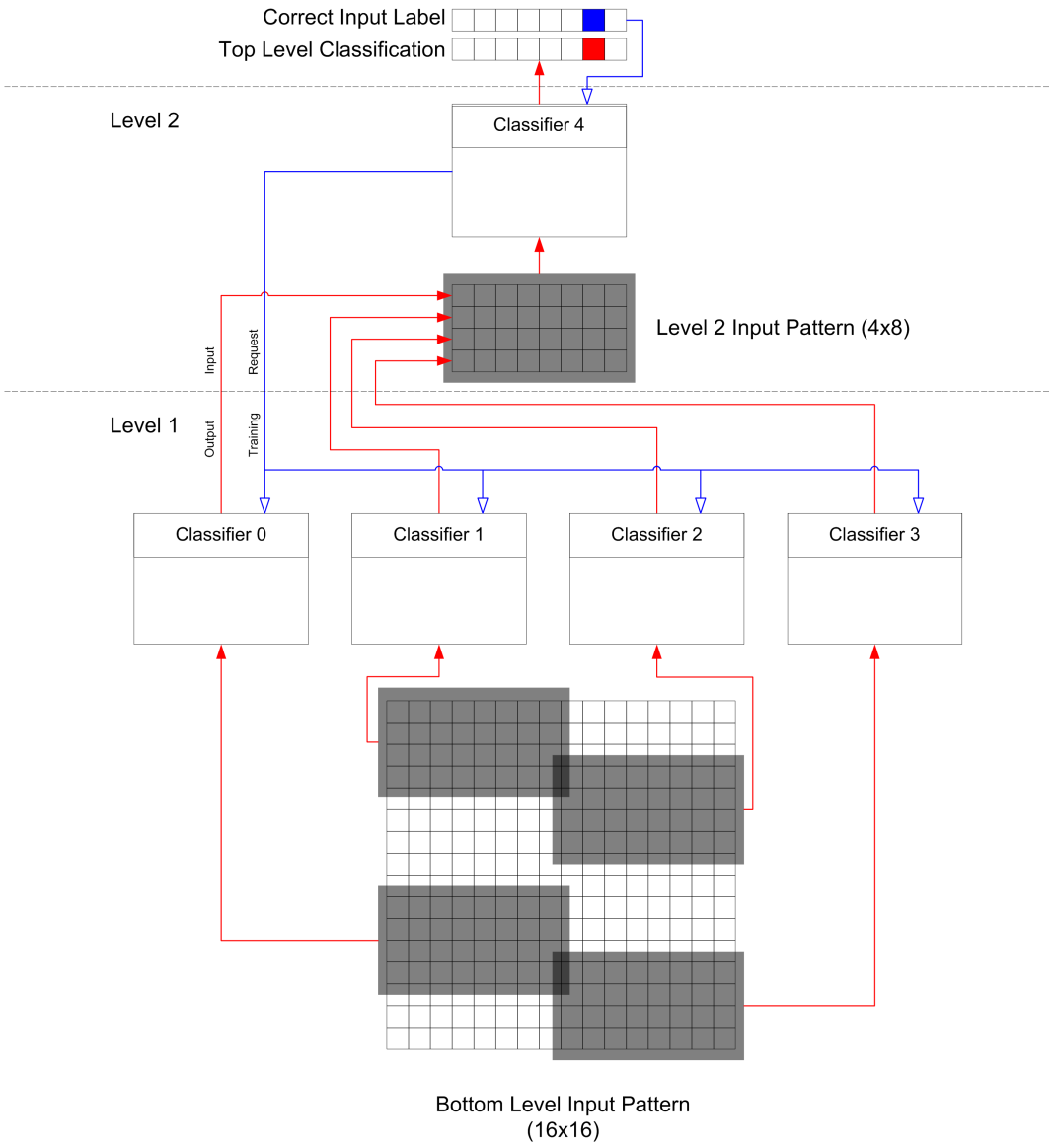


Figure 3.4 An example network configuration, comprising five classifier nodes, illustrating the routing of signals between levels, reconstruction of input patterns at subsequent levels, and *input* selection regions for classifier nodes.

The indices for requesting are based on the order that the classifier nodes are declared in, starting with 0 referring to the bottom left classifier node. The hexadecimal value `0xFFFF` is used to disconnect a request path. More levels can be added, or the array is terminated by a 0.

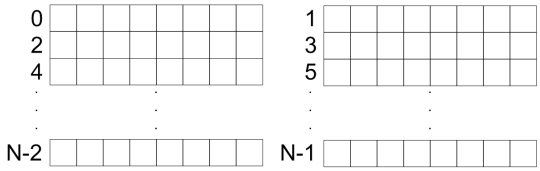


Figure 3.5 Input indices for an N-by-N level input pattern.

3.2.1 Sequence of Events

The sequence of events that occurs in each time iteration in the network is itemised and described in the following. An iteration in the network occurs each time a new input pattern is presented to the network, during training or testing.

1. An input bit-pattern is generated and labelled.
2. The training label becomes the *training* signal for the top classifier.
3. The state of each classifier node is found using the state scripts. Next, the *training* signals propagate through the network from top to bottom, according to the request script instructions.
4. Working a level at a time and beginning at the bottom level, the classifier nodes take their *input* from the level input pattern, determine the quality and order of SOM node matches for their *input*, calculate their new state, then consult the output scripts to determine their output to the level above. This process then repeats in the subsequent higher levels.
5. At the top level, the *output* is the classification for the current input bit-pattern. This is compared to the training label for the current input bit-pattern to see if it has been correctly classified.
6. Finally, the classifiers recalculate their state and consult the learning scripts to determine their learning actions.

The state of the classifier nodes must be recalculated each time the signals are updated as these are a fundamental component in determining their state.

3.3 SCRIPT INSTRUCTION DETAILS

A large space of potential script instructions was created for the GA to select from. The set of script instructions defines the operations which can be performed by the classifier nodes. A discussion is given in the following of each of the types of script instructions: state, learning, output, and request. These are the instruction sets from which the GA could tune the solution to the current classification problem.

3.3.1 State Scripts

State scripts determine what state a classifier node is in. The purpose of state scripts is to allow the network to develop distinct modes of behaviour. Examples of state scripts are presented in the following subsections.

3.3.1.1 Signal Comparison

The following scripts make comparisons with the classifier node's signals.

- `StateAIsElementOfVectorB` - returns true if the signal `A` is an element of signal vector `B`. A signal vector is an ordered list of categories associated with a signal. For example: the signal vector for the SOM winner contains each of the SOM node's indices in order of their match to the current *input*. Tunable thresholds, n, and β are applied, allowing comparison to only the first n signals in vector `B`, or to only signals over a strength β. The signal `A`, and signal vector `B` to be compared can be any combination of the following signals: SOM winner, *training*, and row or column similarity of SOM winner or *training*.

- `StateNoTraining` - returns true if the classifier node received no training from the level above. This script can be used to turn learning off while the classifier is unguided.

3.3.1.2 Internal Parameter Comparison

The following scripts make comparisons with the classifier node's internal parameters.

- `StatePlasticityBelowT` - returns true if the classifier node's plasticity is below some tunable threshold, T.

- `StateConfusionOverT` - returns true if the classifier node's confusion is above some tunable threshold, T.

3.3.1.3 Miscellaneous

- `StateIsTopLevel` - returns true if the classifier node is the top level classifier node in the network.
- `StateSuperTrainingPresent` - returns true if the top level training signal is present.
- `StateTrue` - always returns true.
- `StateFalse` - always returns false.

3.3.2 Learning Scripts

Learning scripts are used to control how the classifier nodes update their learning mechanisms, particularly their SOMs and similarity tables. Examples of learning scripts are presented in the following subsections.

3.3.2.1 Similarity Table

- `AugmentRowAInColB` - incrementes the specified index in the classifier node's similarity table. The row index is specified by a translation of the first parameter, the column index by a translation of the second. The translated indices can select from any combination of the following signals: *training*, SOM winner, *request*, *output*, and row/column similarity of the following: *training*, SOM winner, *request*, or *output*. An example translated instruction might read: "augment row *training* in column winner", where the *training* signal is category 3 and the SOM winner is node 2.

3.3.2.2 Self-Organising Map

In each of the SOM learning scripts the neighbourhood coefficient and learning coefficients are tunable parameters.

- `LearnBestAndNeighbors` - updates the SOM weights according to Equation 2.11.
- `LearnWinnerOverM` - updates the SOM weights according to Equation 2.11, only if the best match is above a tunable threshold, M. A classical SOM implementation would use this script as the learning mechanism with the threshold set to 0.
- `LearnWinnerOnly` - updates the SOM winning node only, allowing forced local learning in the SOM.

- `LearnBestInRangeOrRecruit` - if the SOM winner match is below a tunable threshold, M, then the SOM node which has been the least frequent winner is found, and is treated as SOM winner. The SOM weights are then updated according to Equation 2.11.

- `LearnTraining` - sets the SOM winner equal to the current *training* category and updates the SOM weights according to Equation 2.11.

- `DelearnWorstUnderM` - updates the SOM weights according to Equation 3.1, where the SOM node with the worst match to the current input is treated as the SOM winner. The operation is only performed if the worst match is below a tunable threshold, M.

$$\mathbf{W}(t+1) = \mathbf{W}(t) - N(v,t)L(t)(\mathbf{X}(t) - \mathbf{W}(t)) \tag{3.1}$$

where $W(t)$ is the SOM weights vector,
$L(t)$ is the learning coefficient,
$X(t)$ is the input vector,
$N(v,t)$ is the neighbourhood coefficient.

3.3.2.3 Internal Parameters

- `UpdateConfusion` - decreases, or increases, the classifier node's confusion parameter if the *training* signal does, or does not, match the current *output*. The script uses two tunable parameters, $matchMultiple$ and $mismatchMultiple$, which specify how much to increase or decrease the confusion by.

  Where $1 \leq matchMultiple \geq 100$, $\frac{1}{100} \leq mismatchMultiple \geq 1$.

- `UpdatePlasticity` - decreases or increases the classifier node's plasticity parameter if the *training* signal does, or does not, match the current *output*. The script uses two tunable parameters, $matchMultiple$ and $mismatchMultiple$, which specify how much to increase or decrease the plasticity by.

  Where $1 \leq matchMultiple \geq 100$, $\frac{1}{100} \leq mismatchMultiple \geq 1$.

3.3.2.4 Miscellaneous

- `LearnNothing` - performs no action. This script is used as a place holder instruction, as there can be multiple scripts to execute on each line of the table. Some behavioural modes might only require one learning action, others might require several.

3.3.3 Output Scripts

Output scripts are used to control which *output* bits (catagories) the classifier node will turn on in the *output*. Examples of output scripts are presented in the following subsections.

3.3.3.1 Self-Organising Map

- `OutputBestOverM` - turns on the *output* bit corresponding to the SOM winner node, only if the SOM winner match is greater than a tunable threshold, M. A classical SOM implementation would use this script with the threshold set to 0.
- `OutputAllOverM` - turns on all the *output* bits, corresponding to the SOM nodes, which achieved a match greater than a tunable threshold, M.

3.3.3.2 Similarity Table

- `OutputBestColSimOfWinnerOverM` - turns on the *output* bit corresponding to the column similarity of the SOM winner, only if the similarity is greater than a tunable threshold, M.
- `OutputAllColSimOfWinnerOverM` - turns on the all the *output* bits corresponding to the column similarity of the SOM winner whose similarities are greater than a tunable threshold, M.
- `OutputColSimOfWinner` - turns on the *output* bit corresponding to the column similarity of the SOM winner.
- `OutputRowSimOfWinner` - turns on the *output* bit corresponding to the row similarity of the SOM winner.
- `OutputSignalOverM` - turns on the *output* bit corresponding to any signal with a similarity, or match, greater than a tunable threshold, M. The *output* signal can be chosen from the following: winner, or column/row similarity of winner.
- `OutputWinnerAndColSimOfWinner` - turns on the *output* bits corresponding to the SOM winner, and the column similarity of the SOM winner.

3.3.3.3 Miscellaneous

- `OutputNothing` - turns off all *output* bits.
- `OutputOverlayBits` - separates the 32-bit *input* into four 8-bit segments, then overlays (ORs) the segments to create the *output* bit-pattern.
- `OutputRandom` - turns on a random *output* bit.

3.3.4 Request Scripts

Request scripts are used to control which category or categories are requested from the level below. The *request* signal becomes the *training* signal for the classifier node below. Examples of request scripts are presented in the following subsections.

3.3.4.1 Ideal for Signal

- `RequestIdealForSignal` - requests the bit-pattern which represents the best matching input for a particular node in the SOM. The selected SOM node, translated by a tunable parameter, can be selected from the following: *training*, winner, and column/row similarity of *training*, or winner. An example translated instruction might read: "request ideal for *training*", in which case the classifier will find the SOM node with the same index as the *training* signal, then request the bit-pattern that is the best match to that SOM node.

3.3.4.2 Miscellaneous

- `RequestNothing` - turns off all *request* bits.

- `RequestIsInput` - sets the *request* equal to the current *input*.

- `RequestPassDown` - sets the *request* equal to the current *training*, repeated four times to create a 32-bit vector.

- `RequestBitShift` - sets the *request* equal to the current *training*, repeated four times to create a 32-bit vector. Each of the 8-bit segments is shifted by a tunable parameter, n, number of bits. The idea is to drive the classifiers in different directions in the levels below.

- `RequestRandom` - turns on a random *request* bit, for each classifier currently requesting from.

- `RequestSelectively` - sets the *request* equal to the current *training* but only requests from a selection of the classifier nodes below. For example: in a problem space there are only two possible classification labels at the top level, `A` and `B`. When the *training* signal is `A` the classifier node will request from one half of the possible classifier nodes below only. When the *training* signal is `B`, the classifier node will request from the other half of possible classifier nodes only. A tunable parameter, s, controls whether the operation uses two or four segments.

3.4 GENETIC ALGORITHM

A GA is responsible for searching the script table space and finding the best combinations of scripts and parameters to solve the given classification problem. A population of Dura individuals is maintained, each with its own set of script tables. An overview of the GA search space and the sequence of events which occur during the GA are given.

3.4.1 Search Space

In the GA each individual's chromosome is essentially the four script instruction tables, as described in Section 3.1.2. The size of the script search space, meaning the possible combinations of script `codes` and not including the tunable parameters, can be calculated by Equation 3.2 as given in the following:

$$SearchSpace = \prod_{i=1}^{4} (ScriptSlots_i^{AvailableScripts_i}) \tag{3.2}$$

where the elements of the vector **ScriptSlots** contain the number of script instruction slots to be filled by the four script types: state, output, learning and request; the elements of **AvailableScripts** contain the number of available scripts of the types: state, output, learning and request. The script search space for a typical Dura implementation, with 3 state scripts, 16 learning scripts, 8 output scripts and 8 request scripts is calculated in Equation 3.3.

$$SearchSpace = 3^{16} \times 16^{12} \times 8^{12} \times 8^{12} = 5.72 \times 10^{43} \tag{3.3}$$

When considering the tunable parameters of the script instruction the search space becomes even larger. However, simulations showed that the GA could consistently find good solutions for the tested classification problems in a relatively short amount of time. The longest simulations were run for approximately 24 hours. The simulations also showed that repeated simulations would often find the same or similar solutions.

3.4.2 Sequence of Events

The sequence of events that occurs during the GA simulation are given in the following subsection. Before the GA can begin, the global simulation parameters must be chosen. These parameters are stored by the program in the `EvolveDetails` struct. The C++ code used to implement the `EvolveDetails` struct is given in the following:

```
typedef struct {
    u32 populationSize;
```

```
    u32 superPopulationSize;
    float percentMutate;
    float mutationRate;
    float percentMated;
    float crossoverRate;
    float selectionPressure;
    float survivalRate;
    float spacePresented;
} EvolveDetails;
```

1. **Generate random population**
   `populationSize` individuals are created and given randomly generated chromosomes.

2. **Evaluate fitness function**
   Before the fitness can be evaluated, each individual is trained on the network. For each individual the network is reset and the current individual's scripts are loaded. A number of input samples are presented to the network, selected at random from the input space. The training signal at the top level indicates the correct classification label for the current input sample. The number of samples presented is a percentage of the problem space size and is controlled by the `spacePresented` parameter. The training sequence needs to be long enough to allow the SOMs to settle, and to allow the network to establish its vocabulary between levels. However, training time is also the critical factor in determining the overall speed of the algorithm and therefore needs to be as short as possible while ensuring adequate training. After training is complete the network is tested. During testing a sequence of inputs are presented, with the teaching signal removed from the network. Finally, the fitness of each individual is found from the percentage of input bit-patterns correctly classified. The development of the fitness function implementation is given in Section 5.1.2.

3. **Rank population**
   The population is ranked according to the preliminary fitness assignment, each individual is then assigned a new fitness using non-linear ranking based on the selection pressure, as described in Section 2.3.2.

4. **Create next generation**
   The next generation of individuals is created through the GA mechanisms: elitism, crossover, mutation, spawning, and super-solutions caching. Super-solutions caching is discussed in Section 5.1.3.

   Elitism
   The top individuals are passed unchanged into the next generation. The number

of individuals included in this way is determined by the `survivalRate`.

Crossover

The individuals to be selected for crossover are found using roulette wheel selection based on their fitness, as described in Section 2.5.1. The number of individuals to be generated using crossover is determined by `percentMated`. In each script table, the number of script tokens, and the parameters in the selected script tokens to crossover is found by the `crossoverRate`. The script crossover process is illustrated in Figure 3.6. The offspring produced are then passed into the next generation.

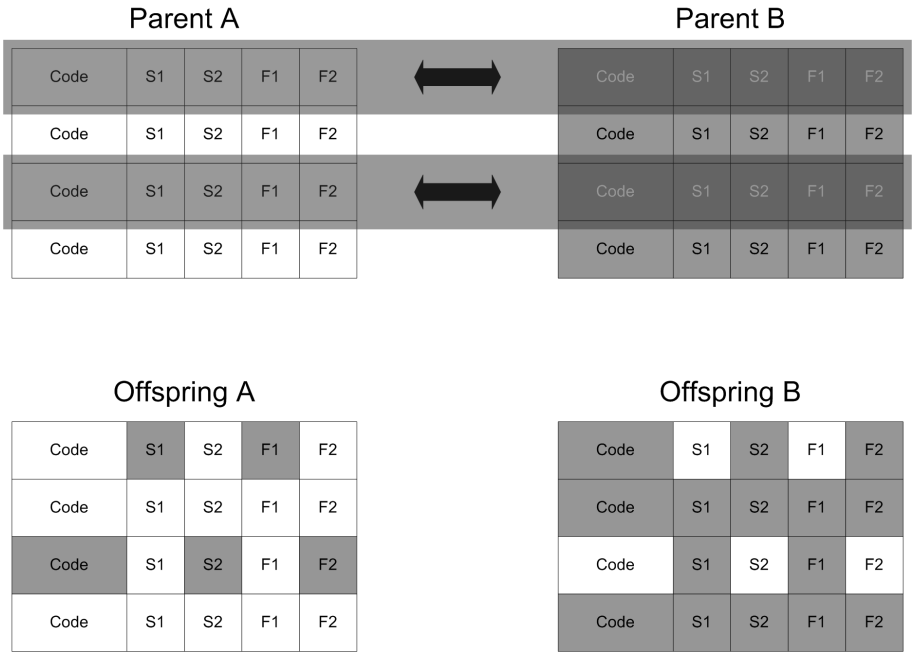


Figure 3.6 Example script table crossover using a `crossoverRate` of 0.5. Each table contains four script tokens.

Mutation

In each script table, the number of script tokens, and the parameters in the selected script tokens, to mutate is found by the `mutationRate`. The selected parameters are then adjusted by a random value scaled by the `mutationRate`. At least one value is guaranteed to mutate during the mutation operation. The offspring produced are then passed into the next generation. The script mutation process is illustrated in Figure 3.7.

Super-Solutions Cache

A random individual is selected from the SSC and inserted into the next generation. The best individual of the current generation then replaces a super solution at a random location in the SSC. The process is discussed in detail in

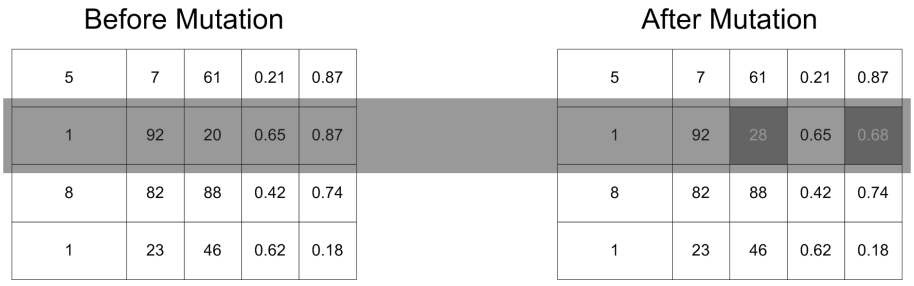


Figure 3.7 Example script table mutation using a `mutationRate` of 0.25. Each table contains four script tokens.

Section 5.1.3.

Spawning
Finally a number of individuals are generated at random in order to bring the next generation population up to the original `populationSize`. If the next generation is too large, the excess individuals can be removed. It is usually desirable to choose the GA parameters such that new random individuals are created in each generation.

5. At the conclusion of the create next generation step, a GA report is created and logged to a text file. The report shows the numbers of, and individuals involved in, each of the GA mechanisms. An example report is given in the following for a `populationSize` of 30.

```
Generation 1092===---
numElitism = 3
numMated = 9 :  (20,28) (23,20) (29,27) (20,15) (6,28) (4,5)
(26,19) (13,26) (20,29)
numMutated = 6 :  3 17 20 19 0 2
numSpawned = 12
```

For each mechanism, the first value indicates the number of individuals, the numbers on the right hand side of the colons indicate the indices of the individuals involved. It can be seen that the higher ranked individuals occur more frequently in the mating group. Individuals are selected randomly for mutation.

6. **Return to Step 2. Evaluate fitness function**
The current generation is complete and the GA returns to step two and repeats until a pre-defined number of generations have been completed.

3.5 PROGRAM OVERVIEW

An overview of the C++ program used to simulate the network is given here. Much care and attention was given to make the C++ code in Dura highly object orientated and easily expandable and maintainable. Several collaboration diagrams were created using Doxygen and GraphViz programs to illustrate the structure of the code. There are two important global class objects in the Dura program, `Network`, and `Evolve`.

3.5.1 Network Class

The `Network` class is the container for all the classifier nodes, and controls the information flow between the classifier nodes. Figure 3.8 shows the collaboration diagram for the `Network` class in the Dura program.

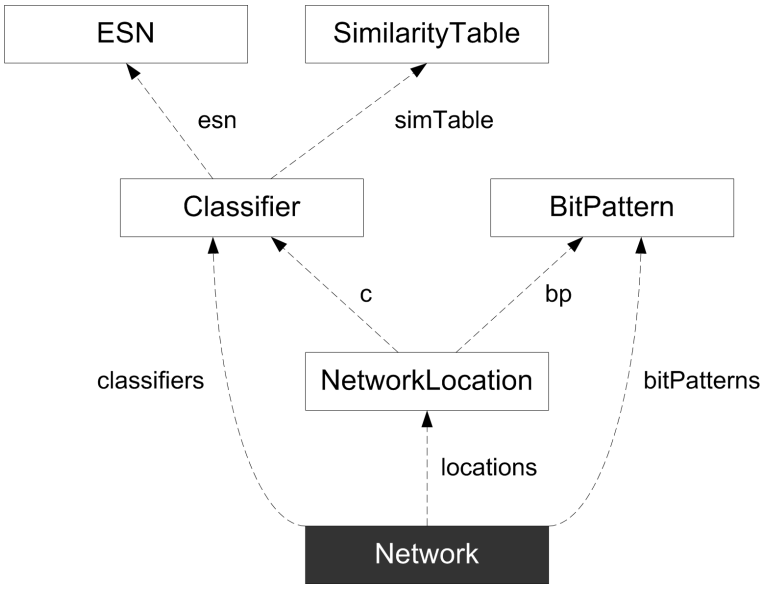


Figure 3.8 Collaboration diagram for the `Network` class.

Figure 3.8 illustrates the basic structure of the `Network` class which contains pointers to the two main elements of the Dura network, the `BitPattern` class and `Classifier` class. The `BitPattern` class implements the bit-pattern inputs seen at each level of the network. The `Classifier` class is the parent class of `Scripted-Classifier` which implements a classifier node. Figure 3.9 shows the collaboration diagram for the `ScriptedClassifer` class in the Dura program.

Figure 3.9 illustrates the use of inheritance to generalise the C++ code and increase the reusability. For example: the virtual class `QVector` is used to implement the `HebbianQVector` class which acts as part of the SOM for a classifier node. The `QVector`

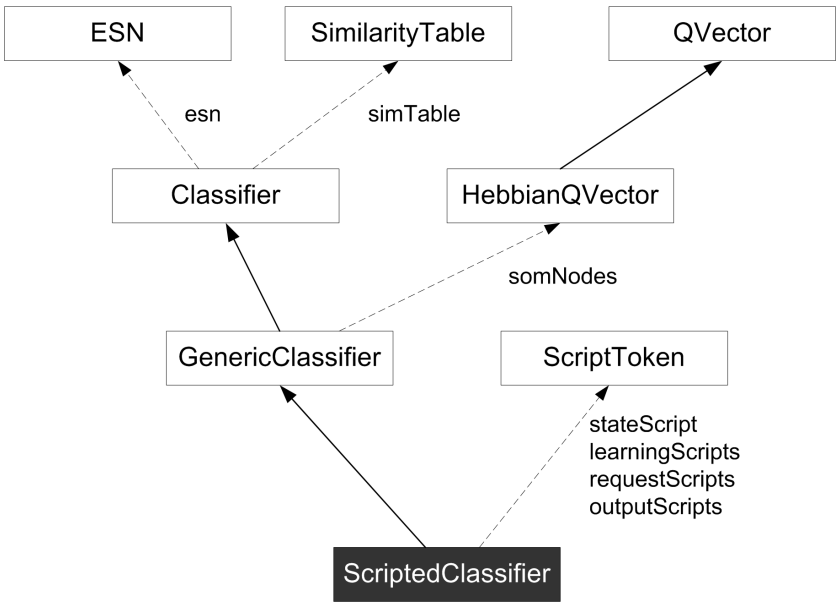


Figure 3.9 Collaboration diagram for the `ScriptedClassifier` class.

class is also used in the code to maintain a record of script activations as seen in Figure 3.10.

3.5.2 Evolve Class

The `Evolve` class is responsible for controlling the GA. The `Evolve` class contains the Dura Individual population and all the evolution statistics and associated data. Figure 3.10 shows the collaboration diagram for the `Evolve` class in the Dura program.

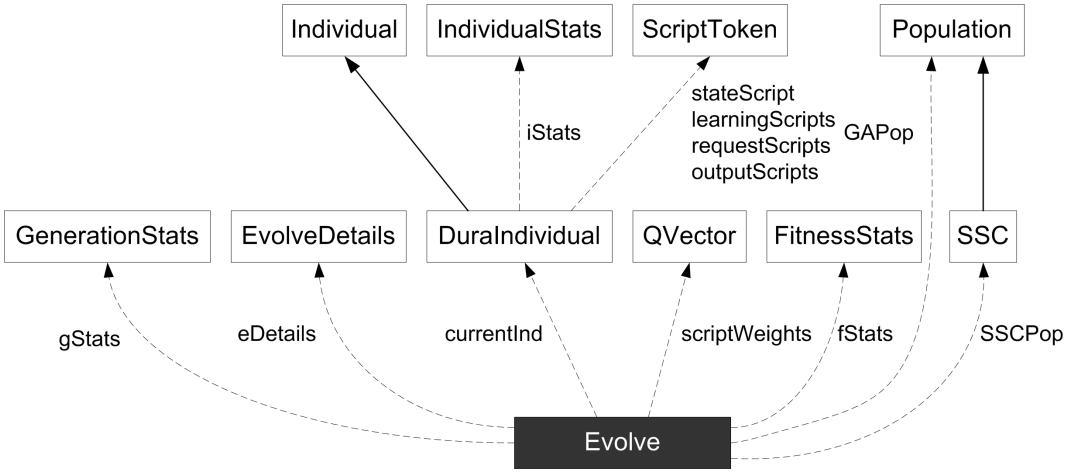


Figure 3.10 Collaboration diagram for the `Evolve` class.

Again, Figure 3.10 illustrates the use of inheritance in the `Individual` and `DuraIndividual` classes. `Individual` contains the virtual functions required to be an individual in a GA (for example, crossover with another individual), `DuraIndividual` implements these functions specifically for a Dura network individual. As previously stated the careful use of object oriented programming principles was an important consideration in the design of this project.

3.6 SUMMARY

The Dura network is a fairly complex system to describe and understand. A summary of the important concepts and design features is given in the following:

- The goal of a Dura network is to correctly classify input bit-patterns from a given classification problem.
- A Dura network consists of a hierarchy of classifier nodes.
- Classifier nodes contain a SOM and similarity table as the main learning mechanisms.
- Classifier nodes communicate to classifiers above and below using four signals: *input*, *output*, *training*, and *request*.
- A classifier node's state is determined by a state script which tests various conditions. Consideration of state takes into account the classifier node's signals, internal parameters, and learning mechanisms.
- Classifier nodes use tables of scripted instructions to determine every action; these include producing *output* and *request* signals as well as updating the learning mechanisms.
- Every classifier node in a Dura network uses the same script tables; therefore, the script tables can be thought of as belonging to the network itself.
- A GA is used to optimize the script tables to solve a given classification problem.
- The GA population is made up of `DuraIndividuals` which are discrete complete Dura networks, each with unique script tables.
- A simulation involves applying a GA to a `DuraIndividual` population to find the best `DuraIndividual` and hence the best script tables to solve the given classification problem.

Chapter 4

EXPERIMENTAL TECHNIQUES

This chapter discusses the experimentation performed with the Dura network in more detail. Given here is a discussion of the user interface used to perform simulations, an overview of the problem spaces tested on the network, as well as a brief overview of the C++ program's structure.

4.1 GRAPHICAL USER INTERFACE

The graphical user interface (GUI) for the Dura project was written with C++ and uses the cross-platform Fast Light Toolkit (FLTK) GUI tools [48]. Figure 4.1 provides a screen shot of the Dura GUI.

1. Input label map for the current problem space - illustrates the correct classification labels for each input pixel on the input map. Each pixel on the 171 by 171 map parses an x and y coordinate to the program, which are used to create the input bit-pattern. Each colour on the map represents a unique 8-bit vector.
2. Output label map for selected classifier - illustrates the selected classifier's outputs for the corresponding inputs.
3. Classifier nodes - this particular network consists of 11 classifier nodes in a three level configuration.
4. Current input bit-pattern - the input bit-pattern to be classified.
5. Level input retina - constructed from the output bit patterns of the classifier nodes from the level below.
6. The top level classification for the current input.
7. The training bit-pattern for the classifier node.
8. The input bit-pattern for the classifier node.
9. SOM weights for the selected classifier node - where white = 1.0, black = 0.0.

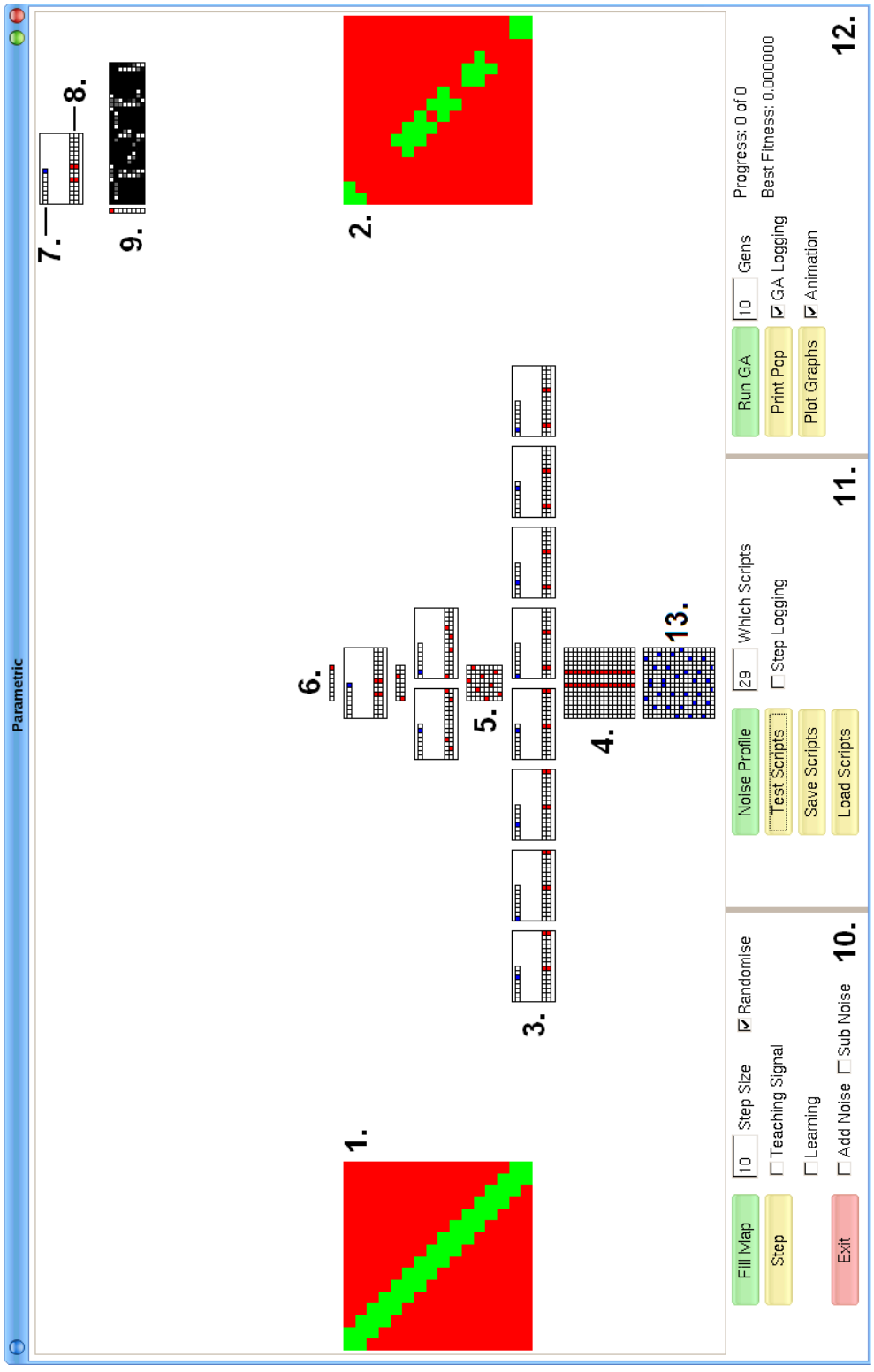


Figure 4.1 Screenshot of the Dura GUI.

10. Network controls - these allow control of various parameters, such as noise levels, and allow batch processing of multiple inputs, at random or sequentially.

11. Analysis controls - these allow testing and noise profiling of selected network individuals from the GA population. These also provide functions for saving and loading of an network individual to a file for future analysis.

12. Genetic algorithm controls - these are responsible for setting the simulation length and starting the GA search. There are also functions for displaying the individuals' scripts, displaying details of the GA population, and producing MATLAB plots for the simulation.

13. Reconstructed Retina - this is constructed from the request bit-patterns of the classifier nodes on the bottom level. Under certain conditions the request bit-patterns would reproduce the input bit-pattern.

4.2 PROBLEM SPACES

The complete set of input bit-patterns, created by the input label map and presented to the network, is called the problem space. For each problem space a bit-pattern generator is used. The generator takes two inputs, an x,y coordinate, and then produces a bit-pattern to be used as the input to the classifier network. Each problem space uses a unique bit-pattern generator. A 2-dimensional input label map can be created for each problem space; this map illustrates the correct label for the bit-pattern that is generated by each x,y coordinate. Initially relatively simple classification problem spaces were presented to the network. After each simulation the results could be analysed to assess where the Dura system was succeeding, or failing, and how the simulations could be improved. An overview of the different types of problem spaces investigated is given in the following subsections:

4.2.1 Exclusive-Or

In the exclusive-or (XOR) problem space there are only 4 possible inputs and 2 category labels, simulating a 2-input XOR logic gate. Figure 4.2 illustrates the input label map as well as a graphical description of the input bit-patterns for the XOR problem space. In the figures that follow, each point on the input label map (left) produces a bit-pattern (right) using a bit-pattern generator. The graphical description of the problem space illustrates the bit-patterns which are possible, usually through a transformation of the example bit-pattern.

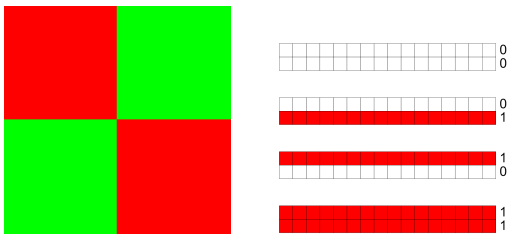


Figure 4.2 (a) Input label map and (b) input bit-patterns for the exclusive or problem space.

4.2.2 Eight Labels

In the 8 labels problem space there are 8 possible inputs and 8 category labels. The 8 labels problem space was designed to investigate the convergence of the SOM to be able to generate 8 unique outputs with only 8 possible inputs. Figure 4.3 illustrates the input label map as well as a graphical description of the input bit-patterns for the problem space.

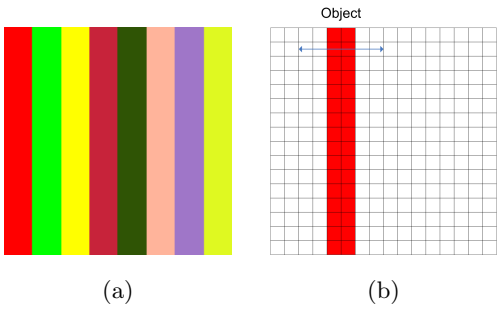
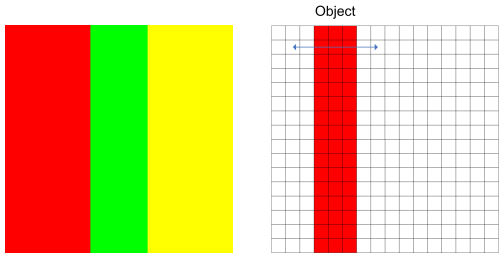


Figure 4.3 (a) Input label map and (b) input bit-patterns for the 8 labels problem space.

4.2.3 Left-or-Right

In the left-or-right problem space there are 16 possible inputs and 3 category labels. The difference between the 8 labels and the left-or-right problem spaces is that in the 8 labels problem space the object can only occur in one of eight locations, two bits apart; in the left-or-right problem space the object can occur centred at 16 different locations. Figure 4.4 illustrates the input label map as well as a graphical description of the input bit-patterns for the left-or-right problem space.



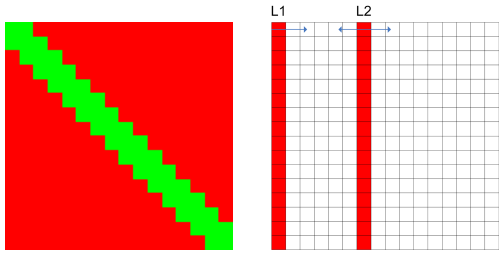
(a) (b)

Figure 4.4 (a) Input label map and (b) input bit-patterns for the left-or-right problem space.

The input patterns are labelled as follows: category `A` if the object appears in the left side of the retina, category `B` if it appears near the the center, and category `C` if it appears on the right.

4.2.4 Overlapping Lines

In the overlapping lines problem space there are 256 possible inputs and 2 category labels. Figure 4.5 illustrates the input label map as well as a graphical description of the input bit-patterns for the overlapping lines problem space.



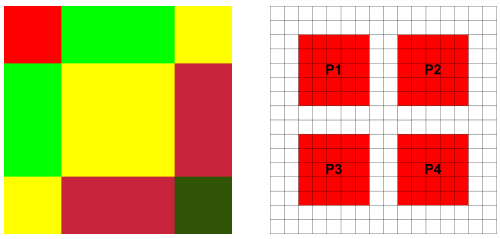
(a) (b)

Figure 4.5 (a) Input label map and (b) input bit-patterns for the overlapping lines problem space.

The input pattern is labelled category `A` if `L1` and `L2` are < 1 pixel apart, otherwise the pattern is labelled category `B`.

4.2.5 Counting Polygons

In the counting polygons problem space there are 16 possible inputs and 5 category labels. Figure 4.7 illustrates the input label map as well as a graphical description of the input bit-patterns for the counting polygons problem space.



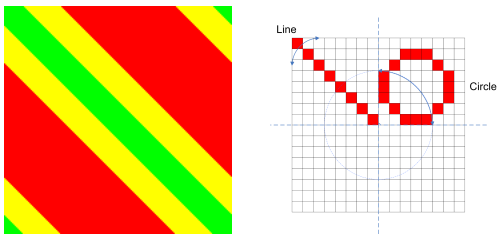
(a) (b)

Figure 4.6 (a) Input label map and (b) input bit-patterns for the counting polygons problem space.

Each of the polygons `P1` to `P4` can be on or off. The input patterns are labelled according to the total number of polygons that were turned on as follows: `A`, `B`, `C`, `D`, or `E` for 0, 1, 2, 3, or 4 respectively.

4.2.6 Line-Circle Overlap

In the line-circle overlap problem space there are approximately 4096 possible inputs and 3 category labels. For this, and other problem spaces, the number of possible inputs is listed approximately due to rounding effects when translating a mathematical orbit onto a 16-by-16 pixel map. Figure 4.7 illustrates the input label map as well as a graphical description of the input bit-patterns for the line-circle overlap problem space.



(a) (b)

Figure 4.7 (a) Input label map and (b) input bit-patterns for the line-circle overlap problem space.

The input pattern is labelled as follows: category `A` if the line and circle are in the same quadrant and not intersecting, category `B` if the line and circle are in the same quadrant and are overlapping, category `C` otherwise.

After early simulations demonstrated that the line-circle overlap was a non-trivial problem space to solve, the problem was broken down into its component parts. Each component part became new problem space and was simulated to gain insight into what steps needed to be taken to solve the line-circle overlap problem. There were four

component problems identified: orbiting point, orbiting circle, rotating line, and dual orbiting points. The component problems are discussed in the following subsections:

4.2.6.1 Orbiting Point

The orbiting point problem space has approximately 64 possible inputs and 4 category labels. Conceptually, the line or the circle, in the line-circle overlap problem space, can be reduced to a single point with some orbit. This problem space tested whether the network could identify which quadrant a single point was in. Figure 4.8 illustrates the input label map as well as a graphical description of the input bit-patterns for the orbiting point problem space.

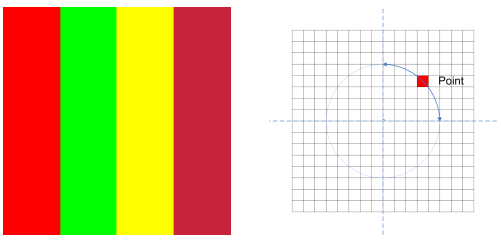
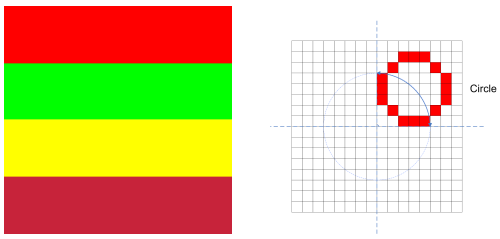


Figure 4.8 (a) Input label map and (b) input bit-patterns for the orbiting point problem space.

The input pattern is labelled depending on which quadrant of the input retina the orbiting point appeared in as either `A`, `B`, `C`, or `D`.

4.2.6.2 Orbiting Circle

In the orbiting circle problem space there are approximately 64 possible inputs and 4 category labels. In the orbiting circle problem the goal is the same as for the orbiting point problem, but the complexity is increased by using a circle, centred at some orbit, rather than a single point. Figure 4.8 illustrates the input label map as well as a graphical description of the input bit-patterns for the orbiting circle problem space.



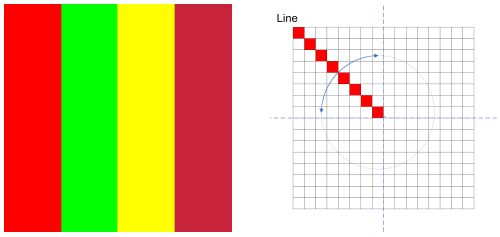
(a) (b)

Figure 4.9 (a) Input label map and (b) input bit-patterns for the orbiting circle problem space.

The input pattern is labelled depending on which quadrant of the input retina the orbiting circle appeared in, as either `A`, `B`, `C`, or `D`.

4.2.6.3 Rotating Line

In the rotating line problem space there are approximately 64 possible inputs and 4 category labels. In the rotating line problem the goal is the same as for the orbiting point problem, but the complexity is increased by using a line, with one end fixed near the centre of the retina and oriented at some angle. Figure 4.10 illustrates the input label map as well as a graphical description of the input bit-patterns for the rotating line problem space.



(a) (b)

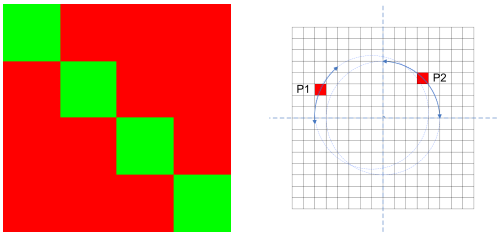
Figure 4.10 (a) Input label map and (b) input bit-patterns for the rotating line problem space.

The input pattern is labelled depending on which quadrant of the input retina the rotating line appeared in, as either `A`, `B`, `C`, or `D`.

4.2.6.4 Dual Orbiting Points

In the dual orbiting points problem space there are approximately 4096 possible inputs and 2 category labels. In this problem space there are two single point objects, each with slightly different orbits. The dual orbiting points problem space tested whether

the network could identify when the two points were in the same quadrant. Figure 4.11 illustrates the input label map as well as a graphical description of the input bit-patterns for the line-circle overlap problem space.



(a) (b)

Figure 4.11 (a) Input label map and (b) input bit-patterns for the dual orbiting points problem space.

The input pattern is labelled category `A` if the two orbiting-points appear in the same input retina quadrant, category `B` otherwise.

4.2.7 Obtaining Data

Important data was logged as tabular text files during simulations, including: script activations of best individuals, fitness statistics, GA mechanisms, and diversity. MATLAB was used extensively in producing plots and visualizations of data obtained. The simulation results for each of the problem spaces is discussed in Section 5.2.

Chapter 5

SIMULATION RESULTS AND DISCUSSION

This chapter discusses specific simulations carried out on the Dura network and gives an analysis of the results. Particular attention is paid to which instruction scripts were important and what roles they played. An investigation into noise performance, solution robustness, and GA generational mechanisms is given. In addition to analysing specific simulations, a discussion of how the SOM and GA were developed during the experimentation process is given.

5.1 ALGORITHM DEVELOPMENT

5.1.1 Self-Organising Map Input Separation

One of the issues with the SOM is that according to Equation 2.10 the input bit-pattern shown in figure 5.1(a) is an equal match (0.75) to the weight vectors shown in Figure 5.1(b) and Figure 5.1(c). However, intuitively it is clear that Figure 5.1(c) is more similar to the input. To try and achieve this distinction the SOM implementation was extended by introducing an additional mechanism to determine the SOM winner using separation.

The separation of the input from the SOM nodes is found by first dividing the bit-pattern into sections, such that each division point is equidistant from the 1 valued input bits. The section start and end indices are stored in the vectors **Start** and **End**. Next the section sums are found by multiplying the weights of the SOM nodes, **W**, by a piecewise linear neighbourhood function, $N(i)$, according to Equation 5.1.

$$SectionSum_j = \sum_{i=Start_j}^{End_j} (W_i \cdot N(i)) \tag{5.1}$$

In each section, the neighbourhood function is 1.0 at the input bit location and decreases exponentially in each direction. Finally the section sums are truncated to $\frac{1}{n}$, where n is the number of sections, and the separation is found by Equation 5.2.

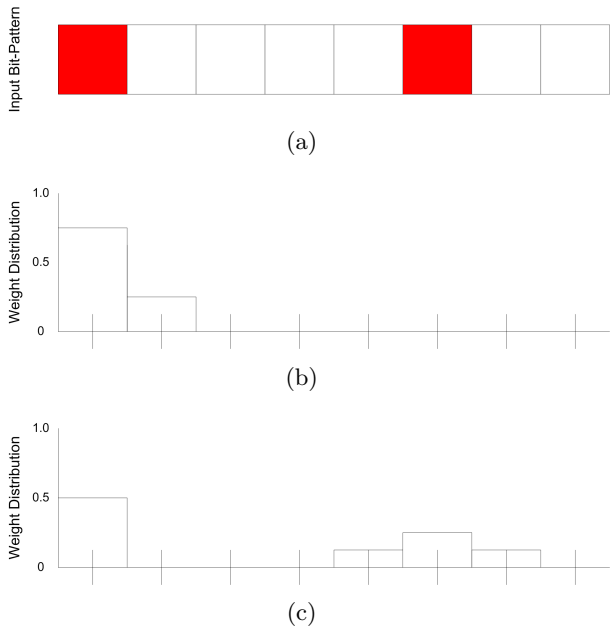


Figure 5.1 (a) Input bit-pattern, (b) an example weight vector with a 0.75 match to the input bit-pattern, (c) an alternative weight vector with a 0.75 match to the input bit-pattern.

$$Separation = 1 - \sum_{j=1}^{n} (SectionSum_j) \tag{5.2}$$

An example 8-bit input bit-pattern and weight vector separation is computed in Figure 5.2, using the same input pattern as that shown in Figure 5.2.

Separation did not replace Equation 2.10 in determining the SOM winner, classifier nodes could implement either or both mechanisms. Scripts that used the separation mechanism include the following: `learnClosestAndNeighbours`, `OutputClosestUnderS`, and `AugmentTrainingInClosest`.

5.1.2 All Red Solution

One of the early problems which arose during Dura simulations was the "all red" solution. The all red solution could occur in a problem space where one category was far more prevalent than any others (for example, the overlapping lines problem). A solution which output the main category every time would still achieve a high fitness. For example: consider a problem space with 95% category A labels, and 5% category B labels. Evaluating the fitness with Equation 2.1 would result in 95% fitness being awarded to a solution that always predicted category A.

This was a significant problem as it could cause the GA to converge on an all red solution too quickly and not allow other, potentially better, genetic material to persist

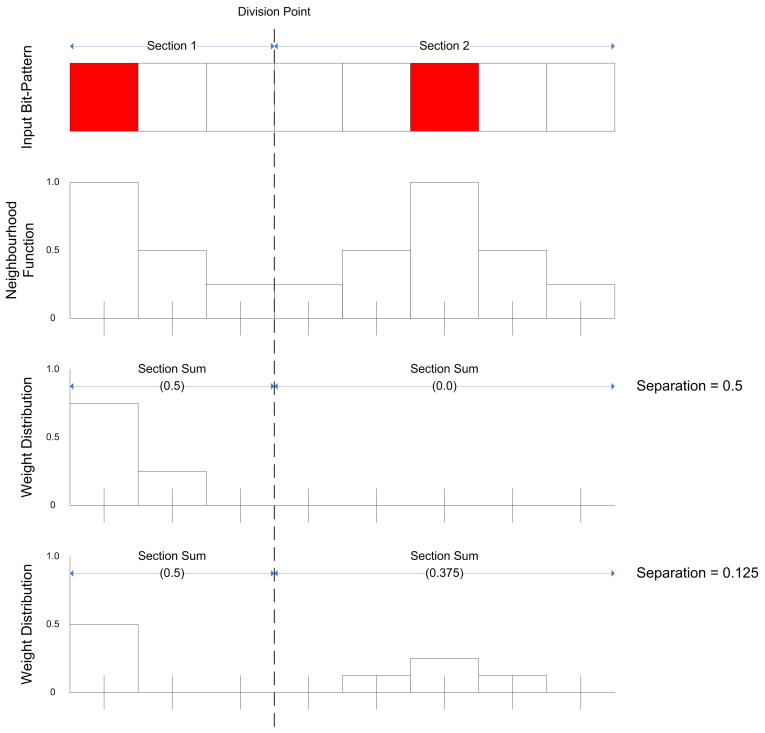


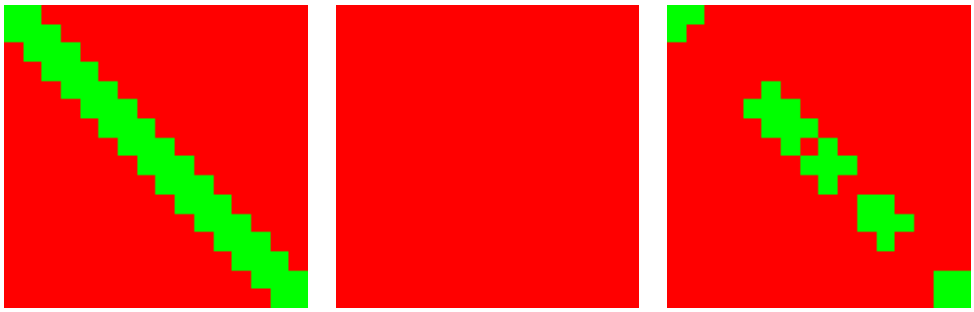
Figure 5.2 Input separation example, illustrating an input bit-pattern, neighbourhood function, and two SOM weight vectors. Division points, sections, section sums, and separations for each weight vector are also shown.

and evolve in the population. To try and alleviate this problem the fitness function was revised to be the average fitness over all categories. The fitness for each category was found then summed and averaged according to Equation 5.3.

$$Fitness = \frac{1}{n}\sum_{i=1}^{n}(\frac{NumCorrectClassifications_i}{NumClassifications_i}) \tag{5.3}$$

where n is the total number of classification categories.

The revised fitness penalises solutions that do not output correctly for all of the categories. Using the new fitness function the maximum value an all red solution classifier could achieve was $\frac{1}{n}$. Consider Figure 5.3 which illustrates the correct output label map and two different solutions' output label maps for the overlapping lines problem. For each output label map the fitness is given using both the old and the new fitness functions.



(a) (b) (c)

Figure 5.3 (a) Correct overlapping lines output label map, (b) all red solution output label map - old fitness = 82.0%, new fitness = 50%, (c) alternative solution output label map - old fitness = 93.4%, new fitness = 81.5%.

Clearly Figure 5.3(c) is a much better solution to the overlapping lines problem than Figure 5.3(b). Under the old fitness function there was only an 11.4% difference in the fitness between these solutions, this represents a 14.0% increase in fitness from Figure 5.3(b) to Figure 5.3(c). Under the revised fitness function the difference in fitness is 31.5%, this represents a 63.0% increase. The greater difference in fitness between the solutions illustrated in Figure 5.3 decreased the likelihood that the GA would converge strongly on the all red solution.

5.1.3 Super-Solutions Cache

One of the problems with a single population GA is their susceptibility to "disasters". A disaster is when the best fitness of a generation drops significantly and does not recover in future generations. This usually indicates that the good solutions in the

population have been lost. With a stochastic training input sample as in Dura, disasters can occur when certain training sequences cause otherwise good solutions to perform badly. Figure 5.4 illustrates the fitness convergence of the GA for a simulation on the left-or-right problem space with no SSC implemented.

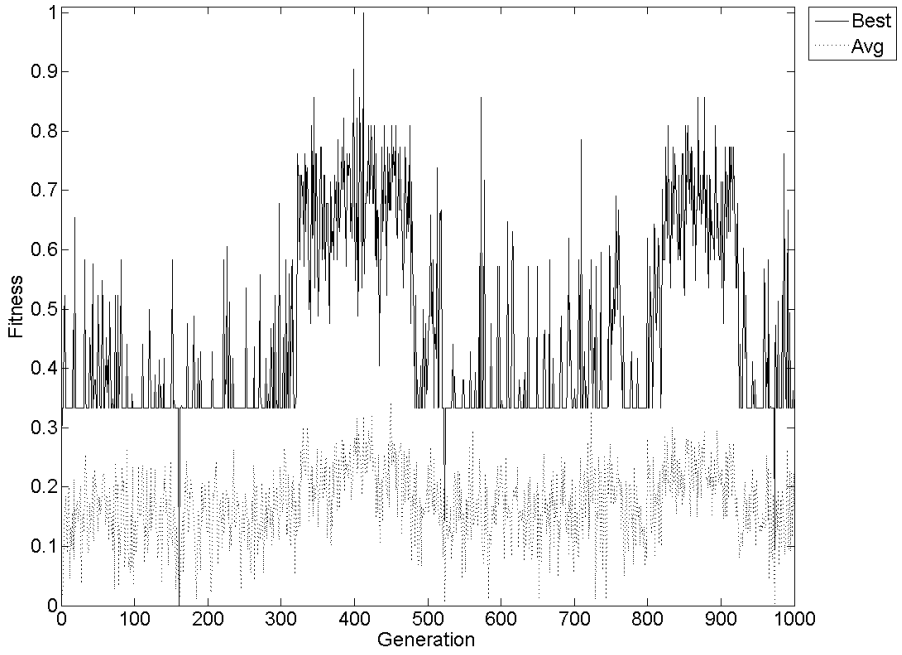


Figure 5.4 Fitness convergence for the overlapping lines problem without a SSC.

In Figure 5.4 the maximum fitness fluctuates from approximately 0.33 to 0.6 until generation 300. It then appears to be climbing until around generation 500 at which point a disaster occurs and the fitness drops to 0.33 again. Another disaster occurs at around generation 900. The GA achieved a fitness of 100% at generation 412. However, the solution was not maintained due to statistical variance in the training samples and the learning.

One of the techniques implemented to combat disasters and prevent loss of good solutions was by a modified form of elitism referred to as the SSC. The SSC is essentially a list of the best individuals from previous generations which can be inserted into the current population. At the conclusion of a generation the best individual replaces a random individual in the SSC. At the beginning of the next generation a random individual from the SSC is injected back into the population. Figure 5.5, illustrates the operation.

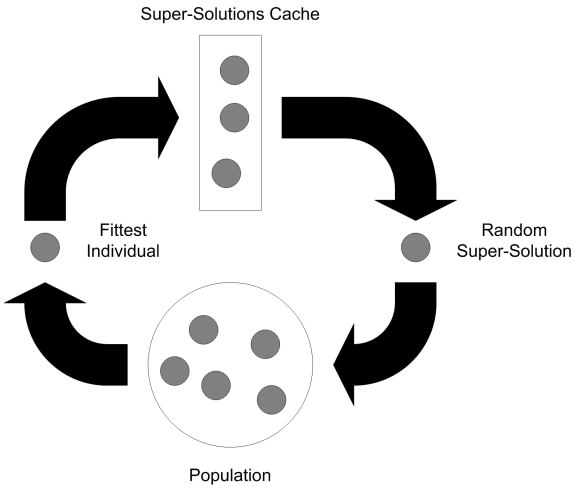


Figure 5.5 Migration of individuals between population and SSC.

The SSC is different from standard elitism in that it can insert individuals from several generations ago into the current generation. Elitism only inserts individuals from the previous generation. In this way, genetic material that may have been useful in the past but then ceased to be useful for a period, can be re-injected into the population where it may become useful again. The SSC acts as a sort of dormant repository of past useful genetic material. The SSC uses a random replacement policy as it is simple to implement and performs relatively well. Other disk cache replacement policies [49] could be implemented to optimise the SSC performance (for example, a first-in-first-out (FIFO) buffer). The overlapping lines simulation was re-run with SSC of size 30. Figure 5.6 illustrates the resulting fitness convergence.

Figure 5.6 shows that there were no disasters in this simulation. There are fluctuations in fitness between generations but in the simulation is far less susceptible to loss of good solutions after their discovery.

5.2 PROBLEM SPACE SCRIPTS

One of the key challenges of this project was to develop tools and visualisations to effectively analyse the problem space simulation results. For each of the problem spaces, discussed in Section 4.2, script activation plots were produced. Script activation plots show how frequently each of the different scripted instructions were used in the fittest individual in each generation of the GA; in these plots, white represents high activation, black represents low activation. Script activation plots allowed analysis of which scripts became more frequently used as a simulation progressed and which scripts were rarely used. By comparing the fitness convergence plots to the script activation plots it was often possible to attribute changes in fitness to the emergence of particular scripts.

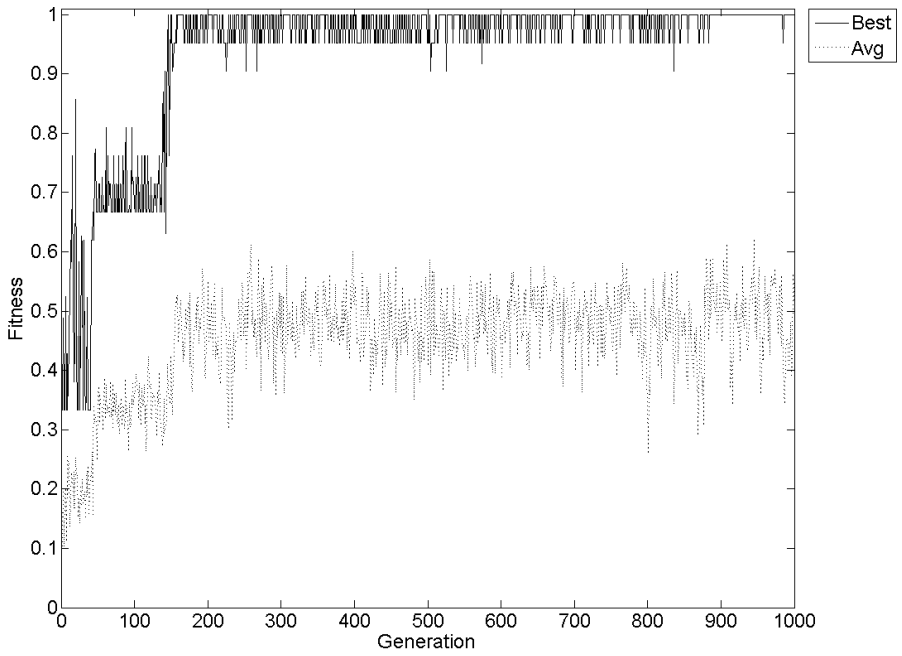


Figure 5.6 Fitness convergence with SSC.

Certain abbreviations are used when labeling the activation plots, for example, "is element of" becomes IEO. It is also important to note that the actual scripts implemented in these plots are not necessarily an exact name match for the descriptions given in Section 3.3. Section 3.3 describes the set of scripts fully, but for each script description there may be a number of actual unique script implementations. For example: an implementation of `StateAISElementOfVectorB` is `StateWinnerIEOTraining`. Each of the problem spaces and the scripts which they employed are discussed in this section.

5.2.1 Exclusive-Or

The XOR problem was implemented as a problem space because it is not a linearly separable problem space. A single boundary cannot be drawn in the input space which separates the inputs which generate a 1 and those which generate a 0. Furthermore a classic ANN cannot solve an XOR without a hidden layer. The XOR problem was a relatively easy problem for the Dura network to solve. The simulation discussed here was performed on a network containing a single classifier node. The solution found demonstrates some of the different ways which the network could behave to solve a classification problem. Figure 5.7 illustrates the fitness convergence for the simulation.

It can be seen from Figure 5.7 that the fitness converged to 100% after only approximately 10 generations. Figure 5.8 illustrates the state script activations and shows that

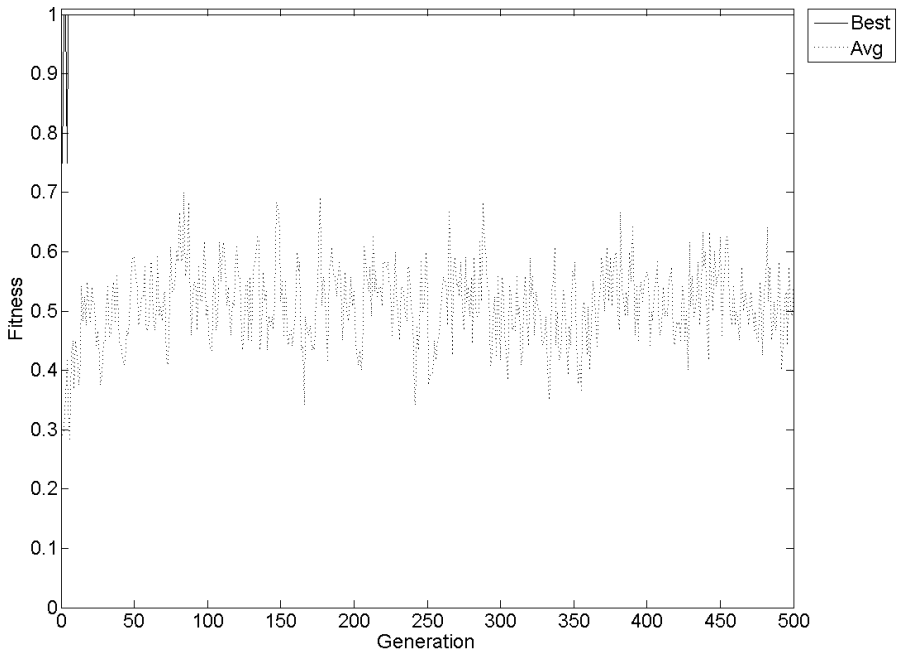


Figure 5.7 Fitness convergence for the XOR problem.

the two most frequently used state scripts were `StateTrainingIEOColSimOfWinner`, and `StateWinnerIEORowSimOfWinner`.

Figure 5.9 illustrates the output script activations and shows that `OutputClosest-UnderS` was the most frequently used output script, with `OutputColSimOfWinnerOverM` also being used.

The request script activations were not relevant in this simulation as the network was only one level. The most frequently used request script however was `Request-Nothing`.

Figure 5.10 shows that there were a number of different combinations of scripts available which could produce a 100% solution to the XOR problem. The main learning scripts however were `LearnBestInRangeOrRecruit`, `LearnClosestOrLeastUsed` and `AugmentTrainingInClosest`. Essentially, there were two distinct ways which the network could solve the problem. The state scripts appeared to be place holders that switched between the different modes described in the following - either of which could solve the problem.

In the first scenario, the similarity table was used to output the column similarity of the SOM winner, which was the most requested category for the SOM winner because of the `AugmentTrainingInClosest` learning script. The SOM itself was updated using either separation or SOP to find the winner. In this way the SOM nodes could converge

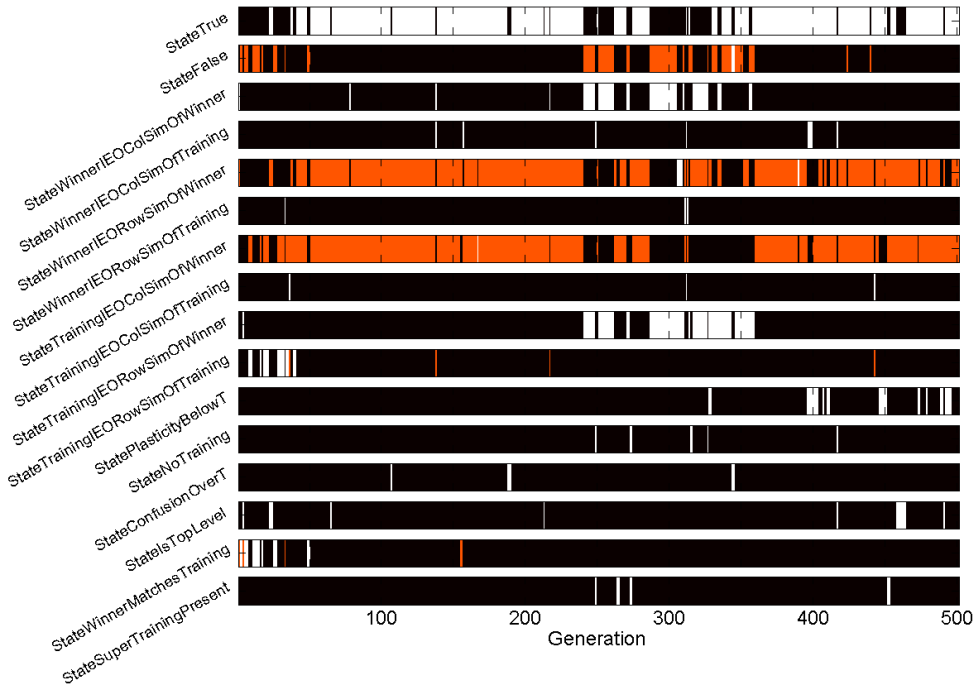


Figure 5.8 State script activations in the best solution in each generation for the XOR problem.

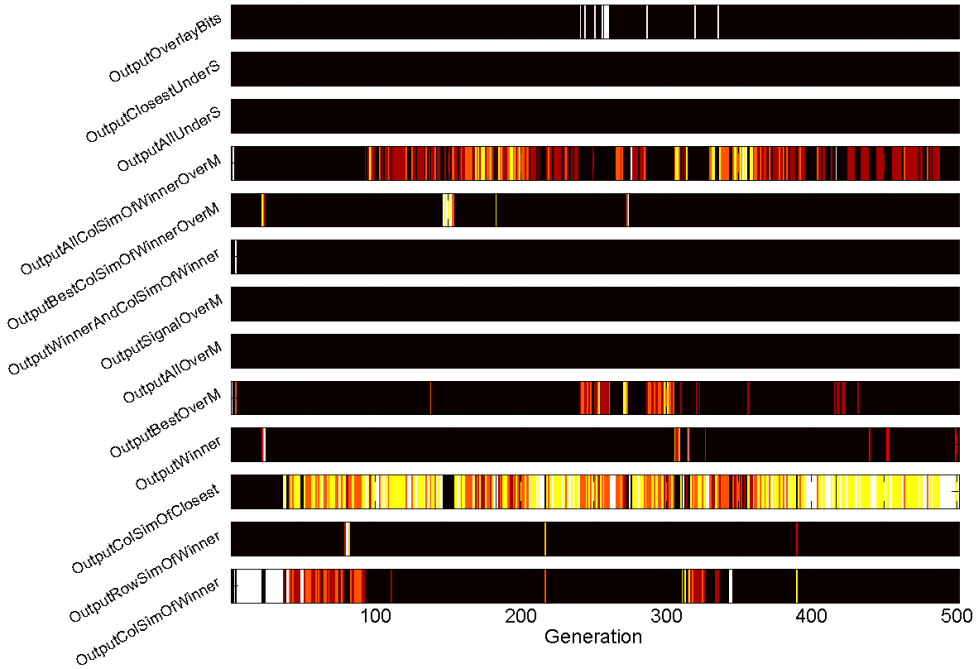


Figure 5.9 Output script activations in the best solution in each generation for the XOR problem.

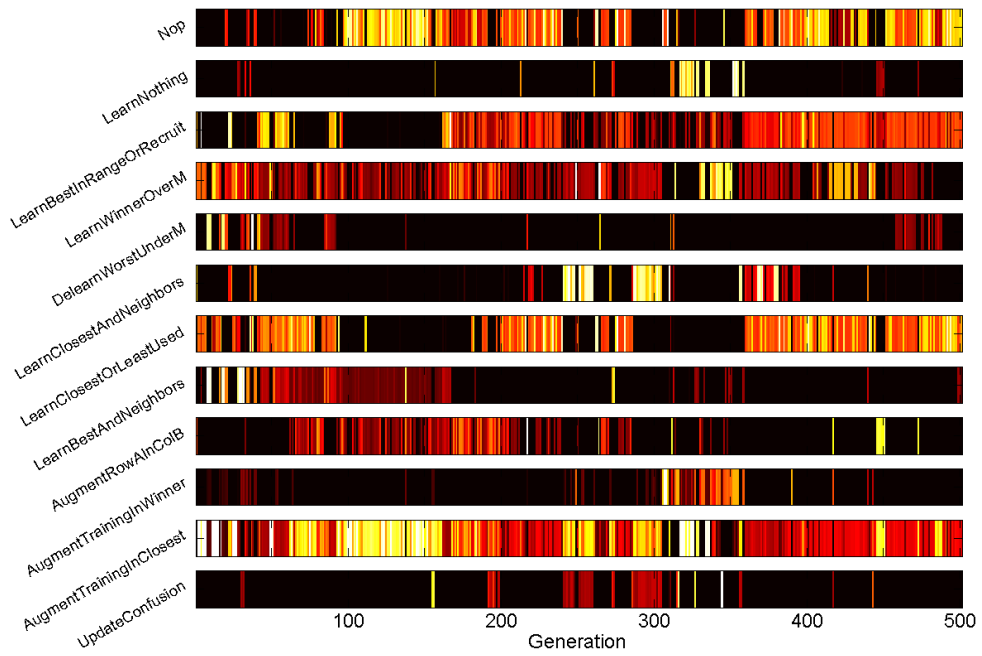


Figure 5.10 Learning script activations in the best solution in each generation for the XOR problem.

on a specific input and then have the output translated to the training category by the similarity table.

In the second scenario, the SOM was updated using either separation or SOP to find the winner. Finally the output was found directly by the SOM winner using separation. The ideal least separate weight vector to the all 0 and all 1 inputs is the same, which is a perfectly evenly distributed weight vector. A second SOM node could converge so that it had a high distribution at the centre of the vector. Therefore, becoming a better match, by separation, for the inputs representing 01 and 10.

In both modes the network could consistently achieve 100% fitness and that is why both behaviours persisted in the GA.

5.2.2 Eight Labels

The eight labels problem space was a relatively simple problem and was implemented with the intention of testing the operation of SOM scripts. A simulation is discussed here using a single classifier node. The simulation reached a fitness of 100% after a single generation. Figure 5.11 illustrates the SOM weights for the classifier node after training. It can be seen that not only have the node's weights adjusted to match the possible inputs exactly, but the nodes also show an ordering of the input. The scripts used to achieve this were `LearnBestInRangeOrRecruit`, and `OutputBestOverM`. This

problem space demonstrates the ability of a SOM to form topologically correct feature maps [50].

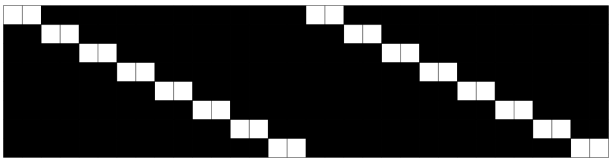


Figure 5.11 SOM weights after training for a classifier node solving the eight labels problem.

5.2.3 Left-or-Right

The left-or-right problem was also a relatively simple problem for the Dura network to solve. A simulation which used five classifier nodes in two levels is discussed in the following. One of the interesting results from this simulation was that there emerged a difference in behaviour between the training and the testing output scripts. Figure 5.12 illustrates the output script activations during testing, Figure 5.13 illustrates the output script activations during training.

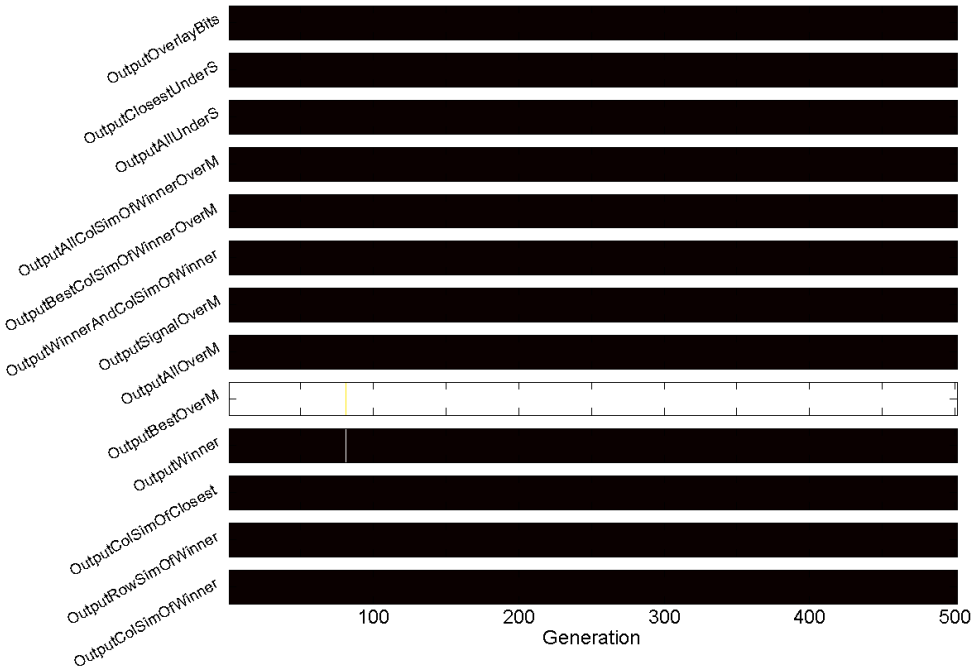


Figure 5.12 Output script activations in the best solution in each generation for the left-or-right problem during testing.

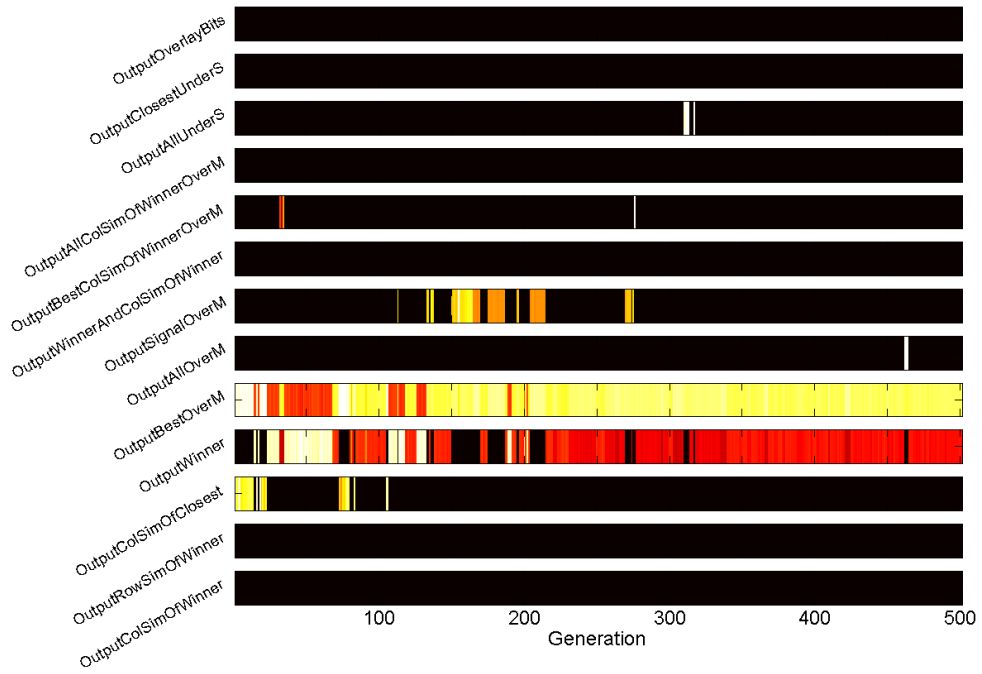


Figure 5.13 Output script activations in the best solution in each generation for the left-or-right problem during training.

The obvious difference between Figure 5.12 and Figure 5.13 is that the scripts `OutputWinner` and `OutputBestOverM` were used during training but only `OutputBestOverM` was used during testing. What this suggested is that during training it was important to output the SOM winner even if it was not a particularly good match to the current input to help promote learning throughout the network. During the testing phase however the SOM winner should only be output if it was above some threshold, causing the classifiers to be more selective and precise.

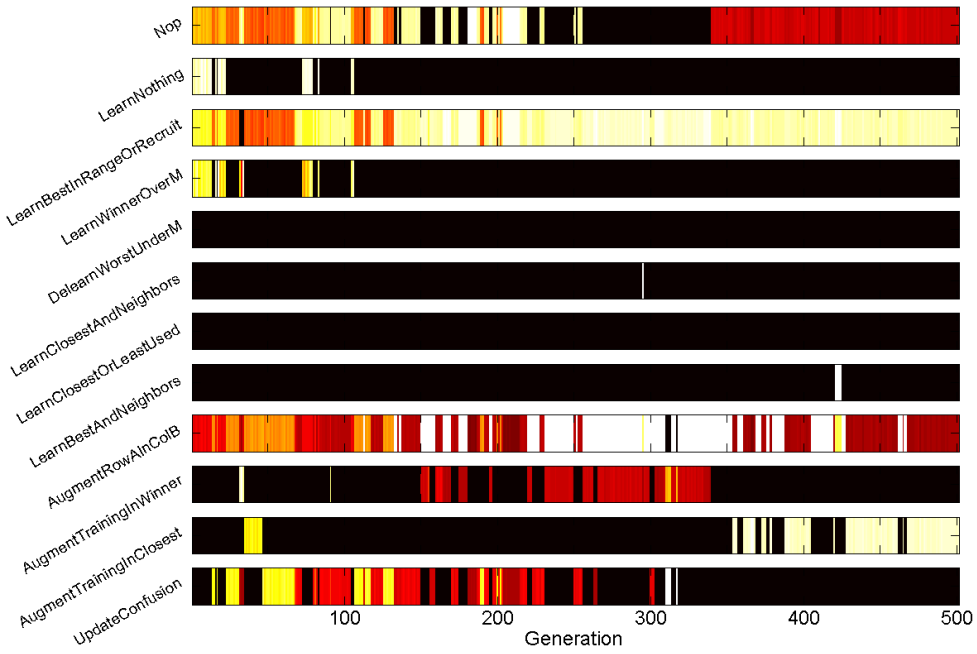


Figure 5.14 Learning script activations in the best solution in each generation for the left-or-right problem.

Similarly to the XOR problem, Figure 5.14 showed that there were multiple combinations of learning scripts which could produce a 100% fit solution.

Observation of the script activation diagrams, which are not all shown for conciseness, and an analysis of the final best solution's scripts revealed that the most important scripts to this simulation were `LearnBestInRangeOrRecruit`, `OutputBestOverM`, `OutputWinner`, and `StateWinnerMatchesTraining`. Essentially the left-or-right problem was being solved using a hierarchical SOM implementation, with the addition of the mechanism to activate `OutputWinner` during training but not during testing. This mechanism was express using the `StateWinnerMatchesTraining` script. Figure 5.15 illustrates the `StateWinnerMatchesTraining` condition for each input step during the solution testing phase.

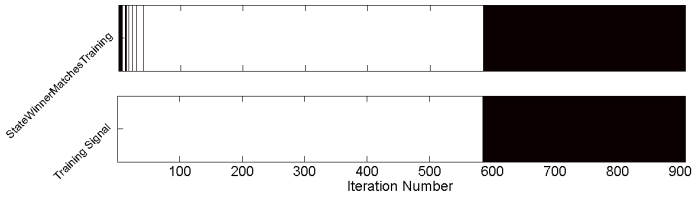


Figure 5.15 Condition of state script `StateWinnerMatchesTraining` as a function of input iterations for the evaluation of a solution to the left-or-right problem.

Figure 5.15 shows that for input samples early in the testing there is disagreement between the *training* signal and the SOM winner, and during this time the `OutputWinner` script was activated. Later when the SOM had settled, during training and testing, `OutputBestOverM` was activated consistently. This simulation showed that the SOM alone could be implemented to solve certain spatial problem spaces.

5.2.4 Overlapping Lines

The overlapping lines problem space proved to be a more challenging problem to solve than originally anticipated. This problem space highlighted the fact that the original SOM mechanism, using SOP to determine the SOM winner, was a limiting factor and led to the development of the separation mechanism. The following discusses an overlapping lines simulation which was performed on a two level network with four classifier nodes in the bottom level and one on the top level. Figure 5.16 illustrates the fitness convergence which reached a maximum of around 96.64%. Before the introduction of separation to find the SOM winner, the overlapping lines problem had been convergeing to around 78% fitness. The following discusses which scripts were used and how the network functioned to achieve the improved fitness.

The state script activation plots for the overlapping lines simulation showed that the three state scripts used almost exclusively throughout the simulation were `ConfusionOverT`, `StateIsTopLevel`, and `StateWinnerIEORowSimOfTraining`.

Figure 5.17 shows that the two output scripts used almost exclusively throughout the simulation were `OutputAllClosestUnderS`, and `OuputColSimOfClosest` . This is an example of a dominant choice in the scripts which is evidence that the selected scripts were particularly important to this problem space.

Figure 5.18 shows that the two request scripts used most frequently were `RequestIdealForWinner`, and `RequestIsInput`. The particular implementation used in the final solution appeared to be bottom-up propagating, and was not dependent on the request scripts.

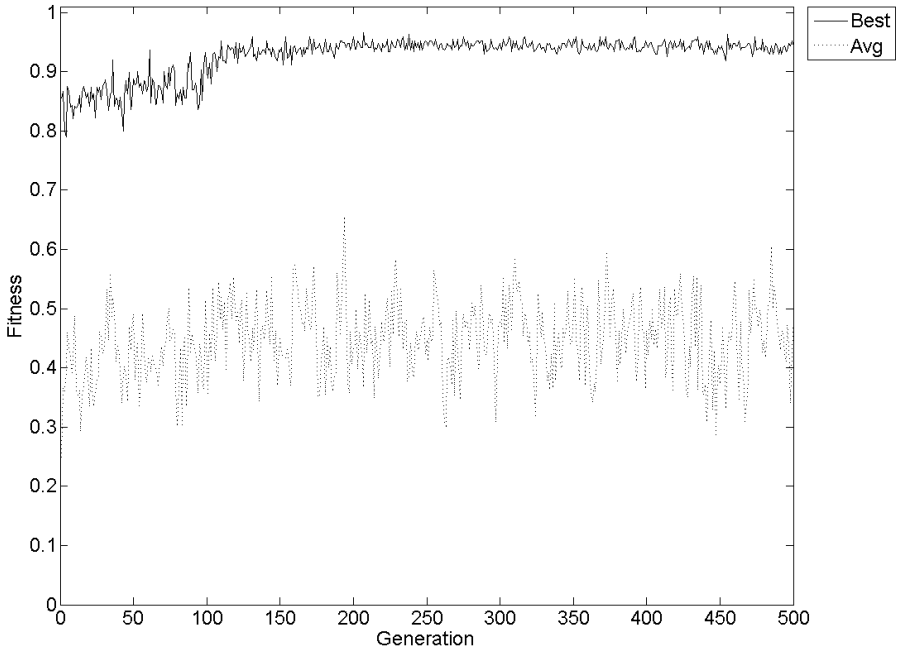


Figure 5.16 Fitness convergence for the overlapping lines problem.

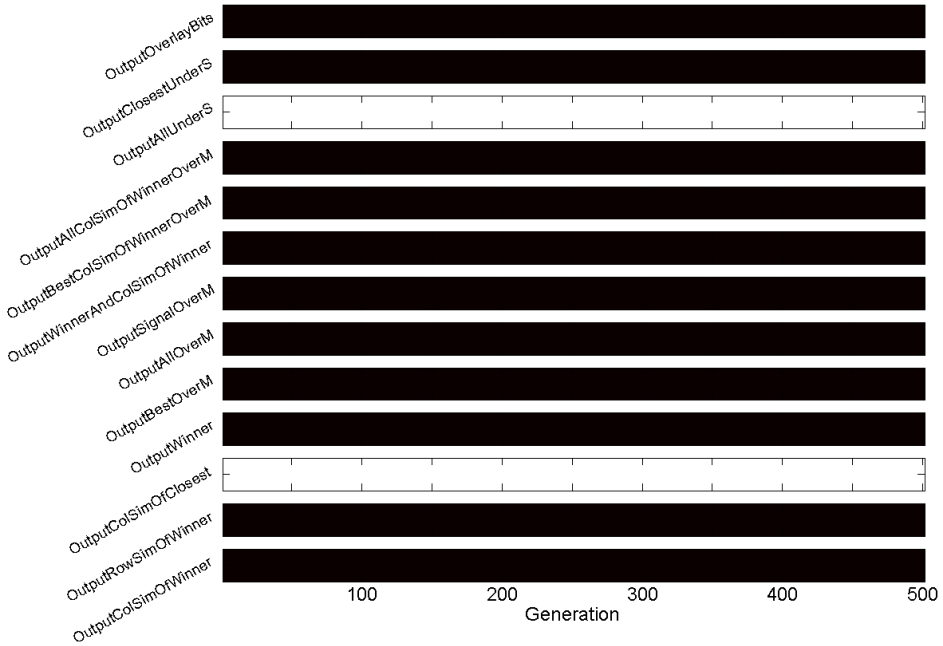


Figure 5.17 Output script activations in the best solution in each generation for the overlapping lines problem.

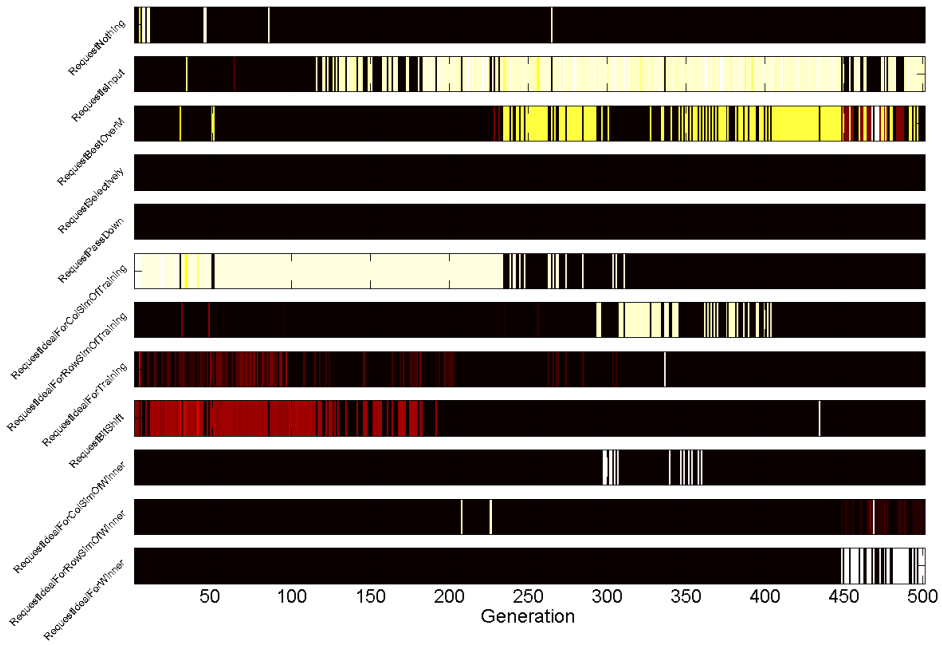


Figure 5.18 Request script activations in the best solution in each generation for the overlapping lines problem.

Figure 5.19 shows that the most frequently used learning script was `AugmentTrainingInClosest`, with `LearnBestOverM` also being used. The demise of the learning script `LearnClosestOrLeastUsed` at around generation 100 appears to coincide with a jump in fitness. The reason for this is explained in the following.

In the final solution to the overlapping lines problem, the top level and the bottom level were performing different tasks. In the top level classifier the similarity table was being used in conjunction with the SOM. The top level classifier was translating the SOM winner (found using the least separation) to match the top level *training* signal using the similarity table. The similarity table was being updated using the `AugmentTrainingInClosest` script and the *output* generated by the `OutputColSimOfClosest`. This mechanism was only expressed in the top level classifier because of the state script `StateIsTopLevel`. Even though the scripts are homogeneous throughout the network, the state scripts allowed classifiers to switch into different modes of behaviour.

The bottom level classifiers were generating their *output* using the script `OutputAllUnderS`. Figure 5.20 illustrates the SOM weights and output label map for a bottom level classifier node.

From Figure 5.20 it is apparent that the SOM weights are fairly evenly distributed. This was achieved by the random generation of SOM weights at the beginning of the simulation. The predominant *output* in the classifier output label map Figure 5.20,

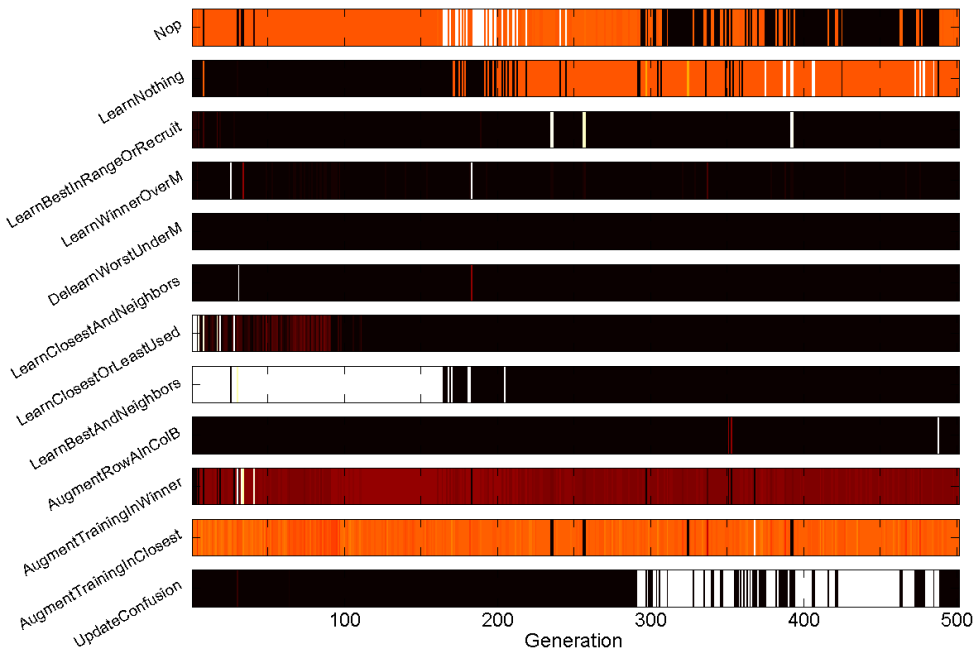
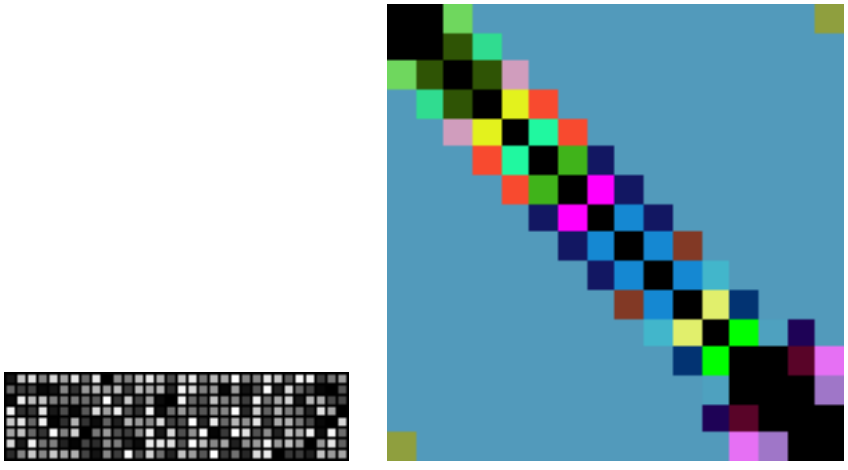


Figure 5.19 Learning script activations in the best solution in each generation for the overlapping lines problem.



(a) (b)

Figure 5.20 (a) SOM weights and (b) output label map, for a bottom level classifier in a network presented with the overlapping lines problem.

is 255 (all *output* bits on). Because of the even distribution of SOM weights, for most *inputs*, each node has a similar separation from the input. Therefore, the most likely *output* is to be either 255, all *output* bits on; or 0, all *output* bits off. As the input pattern lines become closer together, the *input* vector distribution becomes more clustered. Because the *input* is clustered it is less likely to match the SOM nodes as strongly (it is more likely to have a larger separation) and some of the separations will exceed the threshold, S. Therefore, as the *input* becomes more clustered some of the *output* bits begin to turn off, producing the different colours closer to the diagonal in Figure 5.20. Eventually, all of the SOM nodes turned off at the centre of the diagonal in the output label map. This changing *output* provided the top level classifier with distinct *inputs* to translate to the *training* categories with the similarity table as described above.

The reason that the solution did not achieve 100% fitness was because it would output incorrectly when the input pattern lines were displayed on the extreme left and extreme right of the input retina. This was essentially the same effect which caused the changing *outputs* as the *input* became more clustered. When the *input* bits occured at the boundaries of the SOM weight vectors, the separation evalutaion segments were approximately half as big as those for *inputs* which occured away from the boundaries (*cf.* Section 5.1.1). Therefore the separation for those *inputs* was likely to be greater, turning some *output* bits off.

The successful operation of this solution depended heavily on the threshold, S, in the script `OutputAllUnderS`. In situations like this the GA was a powerful tool for optimising the tunable parameters. This solution also depended on a suitable random distribution of SOM node weights. This explains the demise of the `LearnClosest-OrLeastUsed` and `LearnBestAndNeighbours` learning scripts which would change the SOM weights too strongly.

5.2.5 Counting Polygons

The counting polygons problem space is similar to the XOR problem but has increased complexity with 16 inputs and 5 outputs. A simulation was performed using a three level network with 6 nodes on the bottom, 2 in the middle and 1 at the top level. Figure 5.21 illustrates the fitness convergence for the counting polygons simulation. The best fitness achieved was 90% at generation 422. It can be seen from this figure that the best individual's fitness can decrease between generations. This is a result of the stochastic nature of the training inputs. Specifically, the same individual may achieve a different fitness given a different training sequence. Even with elitism SSC strategies in place this effect is noticable. The ability of an individual to acheive consistent fitness over several trials is refered to as it's robustness and is discussed in Section 5.4. The scripts that were used and the behaviours that emerged from the

simulation are discussed in the following.

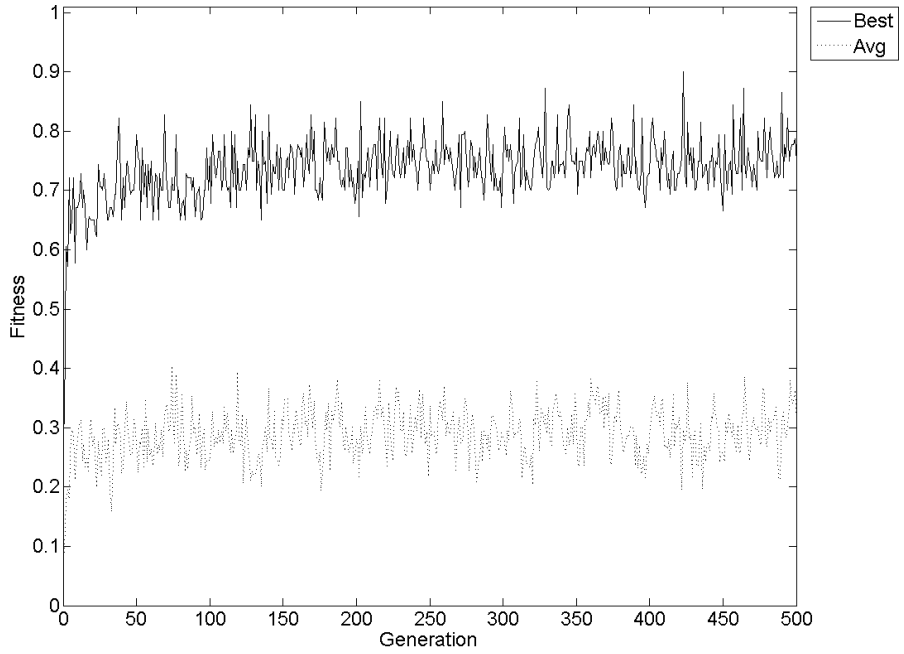


Figure 5.21 Fitness convergence for the counting polygons problem.

The most frequently activated state script was `StateIsTopLevel`, analysis showed that the other scripts were essentially acting as place-holders as they consistently returned the same value during training and testing.

Figure 5.22 shows that the most frequently used learning scripts were `LearnClosestOrLeastUsed`, `LearnClosestAndNeighbours`, and `AugmentTrainingInClosest`. It also shows that `AugmentTrainingInClosest` was activated far more frequently than the other two. Figure 5.22 demonstrates some interesting behaviour at around 100 generations, where the occurrence of `LearnBestAndNeighbours` is phased out and the other SOM separation learning scripts are phased in. This transition coincides with a jump in the fitness convergence from around 70% to around 80%.

Figure 5.23 also shows a change in request script activation at around 100 generations. At this point the emergence of `RequestBestOverM` can be seen. Another interesting observation is that `RequestIdealForColSimOfTraining` receives a high activation up until around generation 390. At this point there is no noticeable change in the overall fitness, suggesting that the particular script was not especially important.

From the instruction script usage the behaviour of the network could be analysed. Essentially this simulation appeared to behave in a similar way to the overlapping line solution discussed above. The actual *request* passed down through the network

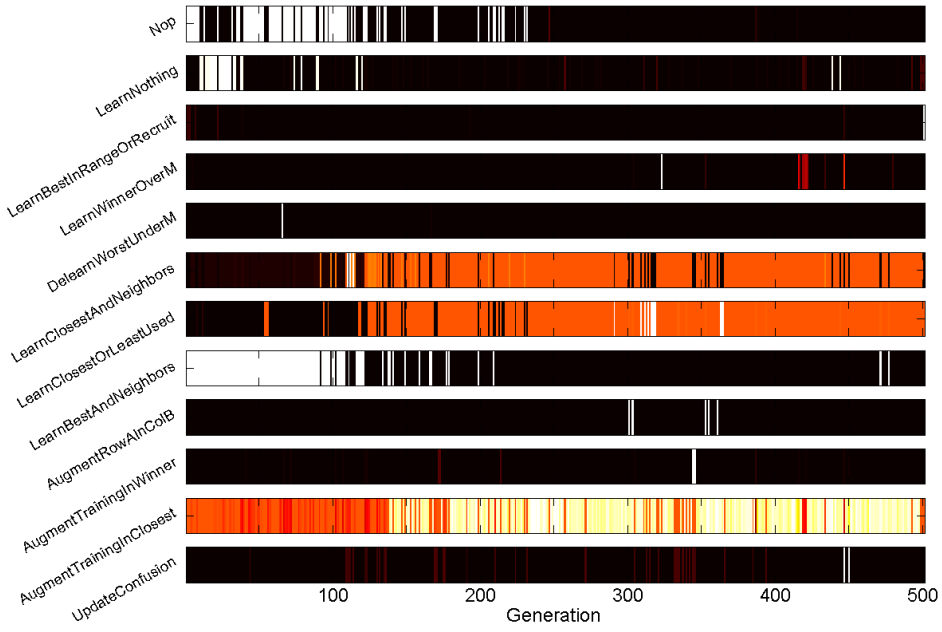


Figure 5.22 Learning script activations in the best solution in each generation for the counting polygons problem.

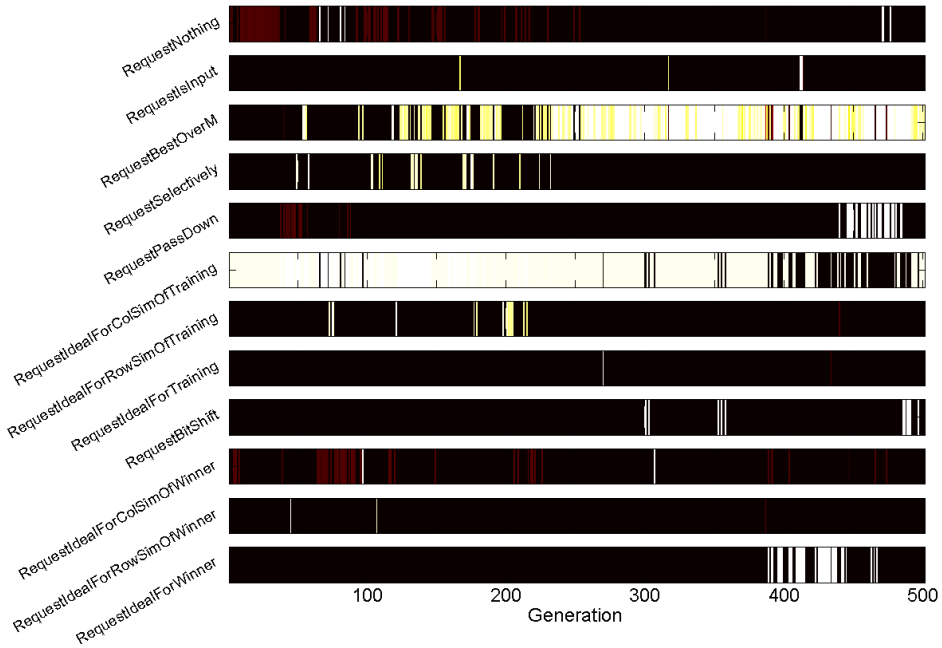


Figure 5.23 Request script activations in the best solution in each generation for the counting polygons problem.

appeared to be non-critical. The *request* could probably have been removed entirely in this simulation. This is because the information being passed up through the network was being controlled by the SOMs, independently of the *training* signal. The top level classifier which was using the similarity table, which is dependent on training, received its *training* directly from the category label. As in the overlapping lines solution the top level was acting as a translator, using the similarity table. The bottom level classifier nodes were being effectively turned on or off as each polygon was turned on or off in the input. The middle level classifiers then had the task of combining the bottom level outputs into meaningful categories. This is where the the network failed to achieve 100%, using separation as the mechanism to determine the SOM winners for the bottom level *outputs* was insufficient. The spacial relationship between the information passed from the bottom level classifiers was too complex to be solved by the eight node SOMs. Under favourable conditions however the network could still achieve a fitness of 90%. The top level classifier output label map for a 90% fit solution is shown in Figure 5.24.

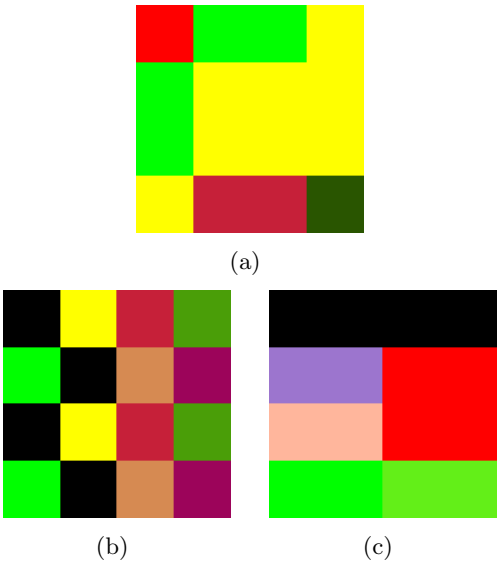


Figure 5.24 Output label maps for (a) the top level classifier, (b) the first middle level classifier, (c) the second middle level classifier in the counting polygons problem.

It can be seen from Figure 5.24 that the network cannot find a relationship between the inputs, generated from the level below, to map the correct outputs. Each of the output labels in the top level map are correct except for the right hand column which have been labelled 2 (yellow) instead of 3 (brown).

5.2.6 Line-Circle Overlap

The line-circle overlap proved to be a non-trivial problem to solve. Early simulations showed this and the problem space was separated into sub-problems: orbiting point,

rotating line, orbiting circle, and dual orbiting points. The aim was to gain insight to each of the sub-problems to determine what was occurring in the line-circle overlap problem. GA simulations were performed on each of the sub-problems and the findings are discussed in the following pages.

The orbiting point, rotating line and orbiting circle simulations all achieved a fitness of 100%. The most important scripts employed in the orbiting point simulation were `LearnClosestAndNeighbours`, `LearnWinnerOverM`, `LearnBestInRangeOrRecruit`, and `OutputAllOverM`.

The most important scripts employed in the rotating line simulation were `AugmentTrainingInWinner`, `LearnClosestOrLeastUsed`, `OutputAllUnderS`, `OutputColSimOfWinner`, and `RequestIdealForColSimOfWinner`.

The most important scripts employed in the orbiting circle simulation were `LearnClosestAndNeighbours`, `StateIsTopLevel`, `OutputAllUnderS`, and `RequestIdealForRowSimOfTraining`.

The most important scripts employed in the dual orbiting points simulation were `LearnWinnerOverM`, `OutputAllOverM`, and `RequestIdealForRowSimOfTraining`.

The orbiting point, orbiting circle, rotating line, and dual orbiting points simulations all exhibited some behaviour similarities. Each classifier node would turn on multiple *output* bits when activated and output nothing when not activated. The classifier nodes were activated by receiving inputs which matched SOM nodes, using either separation or SOP. Regions of activation emerged in the output label maps of the classifier nodes. Figure 5.25 illustrates a trained network presented with the the orbiting circle problem and shows the *outputs* propagating up through the network. Figure 5.26 illustrates the SOM nodes and output label maps for the classifier nodes on the top two levels of the network.

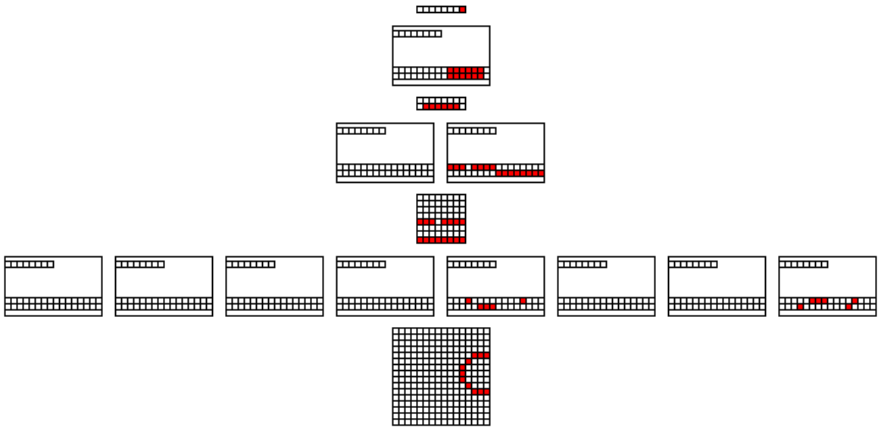


Figure 5.25 A trained network presented with the the orbiting circle problem space.

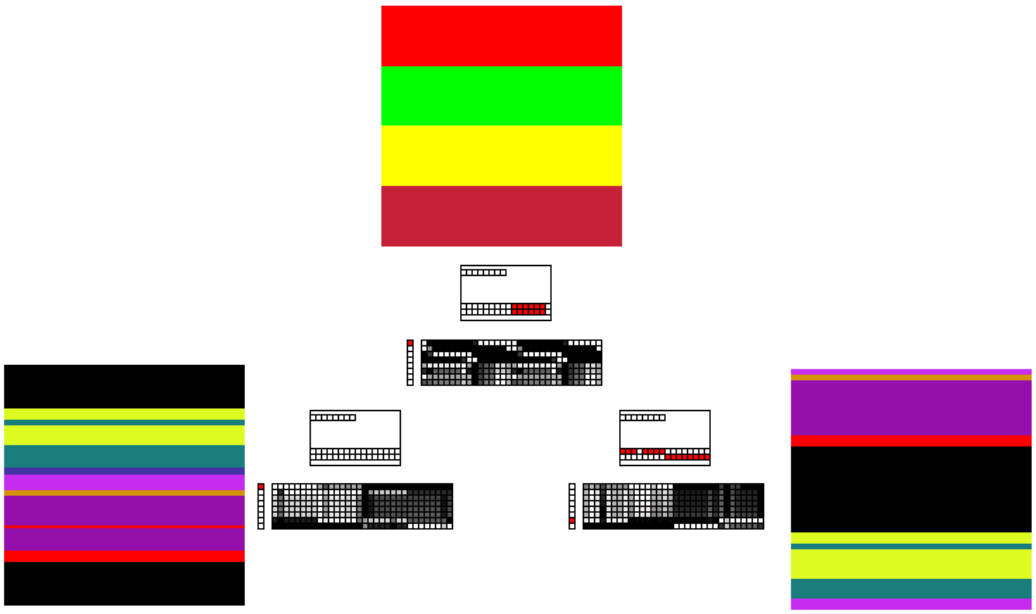


Figure 5.26 SOM weights and output label maps for the classifier nodes on the top two levels of a trained network presented with the orbiting circle problem.

Figure 5.26 shows how regions of activation have emerged in the output label maps of the middle level classifiers. The outputs produced in these regions were then translated by the top level classifier using the similarity table to generate the correct outputs. The fact that the orbiting point, rotating line, and orbiting circle simulations all achieved a fitness of 100% showed that increasing the complexity from a point to a circle or a line was not critical in causing the network to fail.

However the dual orbiting points simulation did not perform as well, only achieving a fitness of 83.06%. This may have been because in the previous simulations only one classifier node was activated at a time. In the dual orbiting points problem the top two levels were receiving input from more than one source at the same time which in turn made the SOM more difficult to interpret at each level.

There were slight differences between each of the solutions found for the sub-problems. It may be these subtle differences which mean that the combination of them all can't occur or is much harder for the GA to find.

Despite the success of the majority of the sub-problems, the line-circle overlap problem itself only achieved a fitness of around 75%. Figure 5.27 illustrates the fitness convergence of the line-circle overlap problem.

The most important scripts employed in the line-circle overlap simulation were `StateIsTopLevel`, `OutputAllOverM`, and `LearnBestAndNeighbours`. The jump in fitness observed at around generation 100 was due to the emergence of the `OutputAll-`

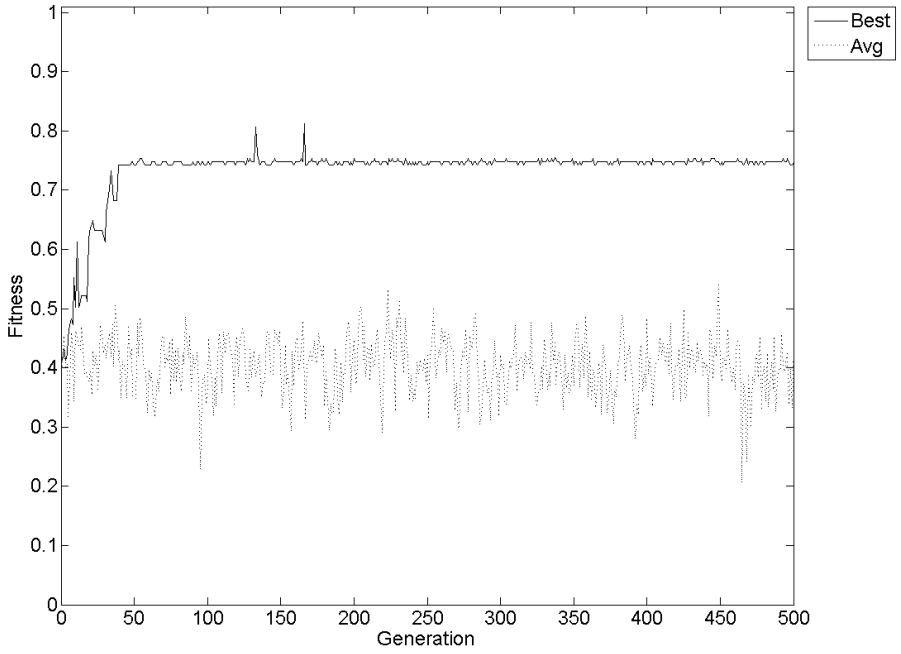


Figure 5.27 Fitness convergence for the line-circle overlap problem.

`OverM` script. The `OutputAllOverM` script uses the SOP mechanism to find the SOM winner not separation. This is similar in effect to the classifier node creating a list of inputs to which it will generate output. For the orbiting point, orbiting circle, and rotating line sub-problems this script was sufficient to create the correct output. However, the much greater input space size of the dual orbiting points and the line-circle overlap problem mean that the SOM nodes could not store enough information to generate the correct outputs.

5.2.7 Performance Summary

The best fitness achieved for each of the problem spaces is summarised in Table 5.1; it shows that the Dura network was successful in classifying most of the problems tested here.

One of the interesting and common behaviours that emerged during simulations was the turning on of multiple *output* bits rather than just single categories. In some cases generating denser *outputs* than *inputs*.

Another common feature was the use of the `StateIsTopLevel` script in combination with SOM learning and output scripts in the bottom level classifiers and similarity table learning and output scripts at the top level. This behaviour allowed density esti-

Table 5.1 Summary of best fitness achieved during simulations for each problem space.

Problem Space	**Best Fitness**
XOR	100.00%
Overlapping Lines	96.64%
Eight Labels	100.00%
Counting Polygons	90.00%
Left-or-Right	100.00%
Orbiting Point	100.00%
Rotating Line	100.00%
Orbiting Circle	100.00%
Dual Orbiting Points	83.06%
Line-Circle Overlap	81.20%

mation to be performed on the inputs and then a translation of the internal signals to the training labels.

Another general observation was that the similarity table was almost exclusively used to augment the requested category in the SOM winner category. This suggests that the other potential similarity table interactions (for example, `AugmentRowWinner-InColSimOfWinner`) were either too complex to be likely to be selected by the GA, or were not particularly useful in solving the types of problems discussed here.

Although there were many similarities in the types of solutions found for the different problem spaces, there were also many differences. There were no underlying script architectures that could solve all the problems tested here. It can therefore be concluded that creating a super-set of scripts from which a GA can tune the optimal scripts for a specific problem is a useful and powerful tool.

5.3 PERFORMANCE WITH NOISE

The performance of solutions in the presence of additive noise was investigated. Additive noise was simulated by adding randomly located bits to the input retina. The noise level was a percentage of noise bits of the total input retina space. Clearly for input patterns that are very sparse, a low noise level is going to be more significant than for an input pattern that is very dense. However, for most problem spaces input pattern densities are similar or the same for all inputs, therefore comparison between solutions for the same problem space with the same noise levels are appropriate. Figure 5.28 illustrates a sample retina for the left-or-right problem in the presence of 5% additive noise.

Consider Figure 5.29, which illustrates the fitness performance with additive noise for five different solutions to the left-or-right problem space. Each of the solutions were

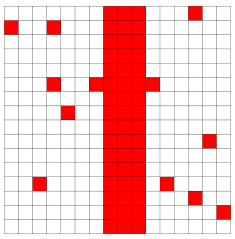


Figure 5.28 Sample retina for the left-or-right problem in the presence of 5% additive noise.

found in simulations with levels of input noise applied during training as follows: 0%, 1%, 5%, 20% and 50%.

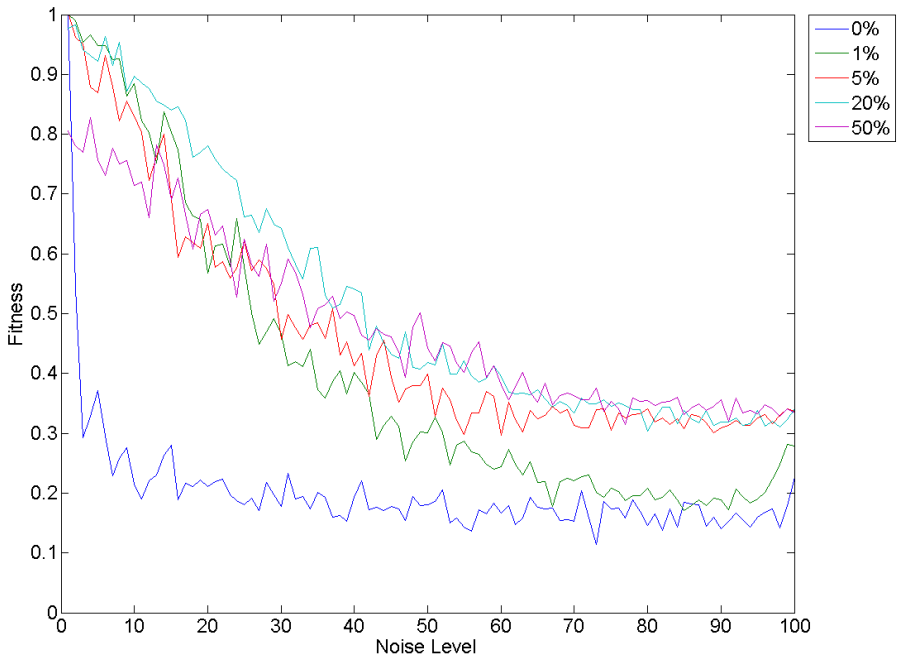


Figure 5.29 Noise comparison for five different solutions to the left-or-right problem space. Each of the solutions were found in simulations with different levels of input noise during training.

The data in Figure 5.29 was generated by training and testing each solution 10 times, then taking the average fitness for each percentage increase of noise. It can be seen that the solution which was evolved without noise, performs poorly during testing with only a small amount of added noise. Even with 1% noise during evolution there is a vast iprovement. 5% and 20% showed improvement again over the 1% solution, especially after around 50% testing noise level. This result shows that the performance of a Dura network solution in the presence of noise can be imporved by exposing to noise at the evolution stage.

If the evolution noise was continuously increased there was no significant increase in performance at the high testing noise levels, but there was a reduced performance at the low testing noise levels. This is most likely due to noiseless inputs being vastly different to any inputs with 50% additive noise seen during the evolution phase.

The next step was to investigate what differences in the solutions brought about the differences in additive noise immunity. Figure 5.30 shows the sum of output script activations in the best solution over the entire simulation. Figure 5.31 shows the output script activation in the best solution at each generation.

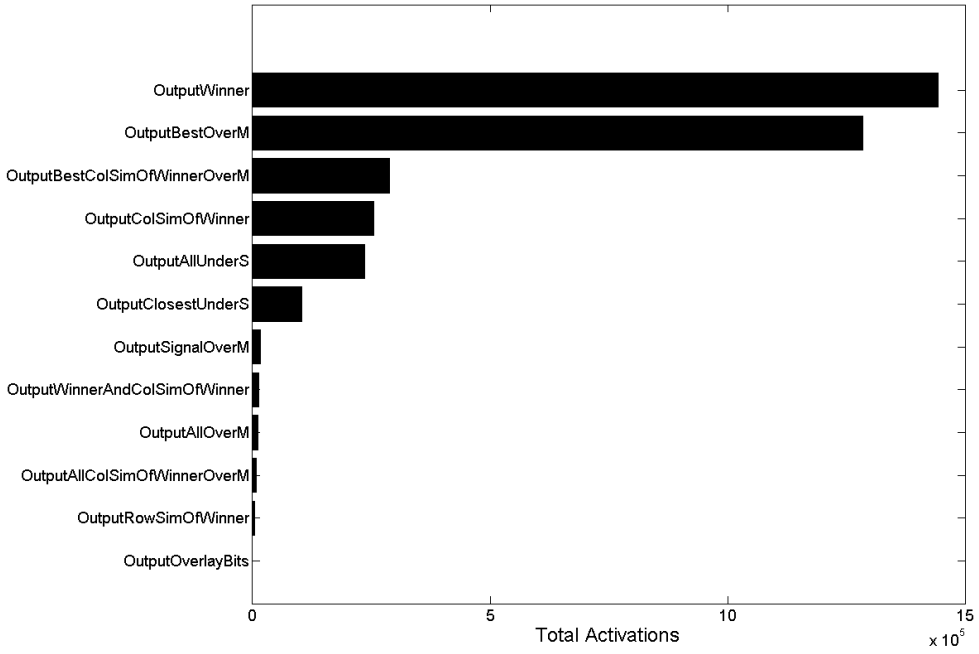


Figure 5.30 Sum of output script activations for the left-or-right problem in a noiseless simulation.

By looking at the output script activation plots it can be seen that the first solution, trained without noise had a relatively even distribution of activations between `OutputWinner` and `OuputAllOverM`, with a significant contribution from three other scripts. This suggests that there were several paths that could be taken to achieve 100% output success for the left-or-right problem space in noiseless environment. The second individual, trained with 1% noise, converged more strongly to using the `OutputWinner` instruction. Figure 5.32 shows the sum of output script activations over the entire simulation. Figure 5.33 shows the script activation in the best solution at each generation.

In the 1% noise environment the `OutputAllOverM` script is practically unused. Using the `OutputWinner` script may increase the noise immunity compared to `OutputAllOverM` for two reasons; firstly, the added noise may reduce the best SOM match below the threshold M, causing the classifier node to output nothing, even if the best

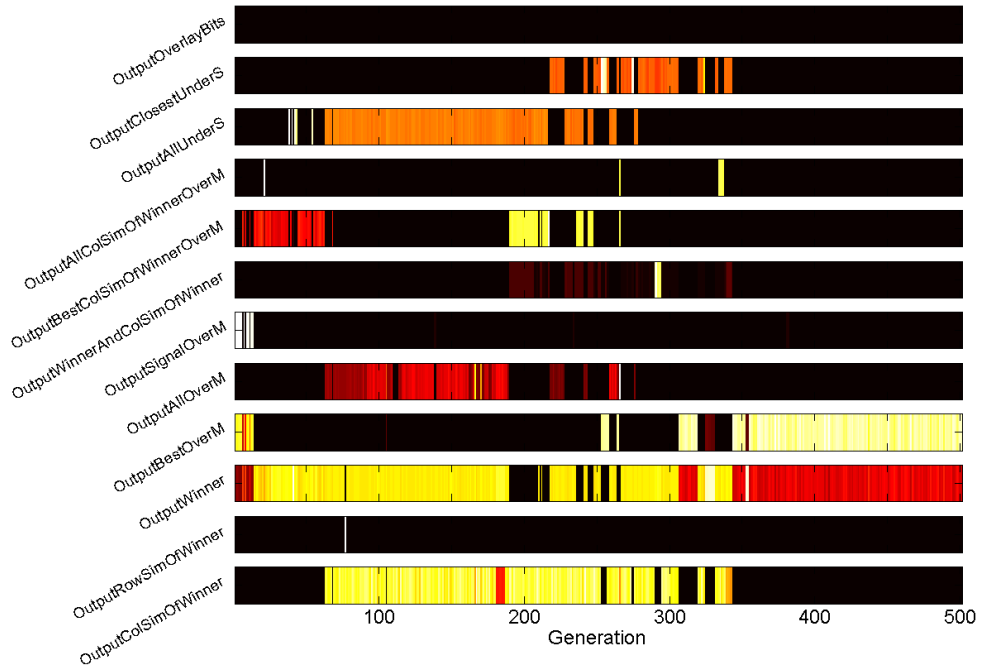


Figure 5.31 Output script activations in the best solution in each generation for the left-or-right problem in a noiseless simulation.

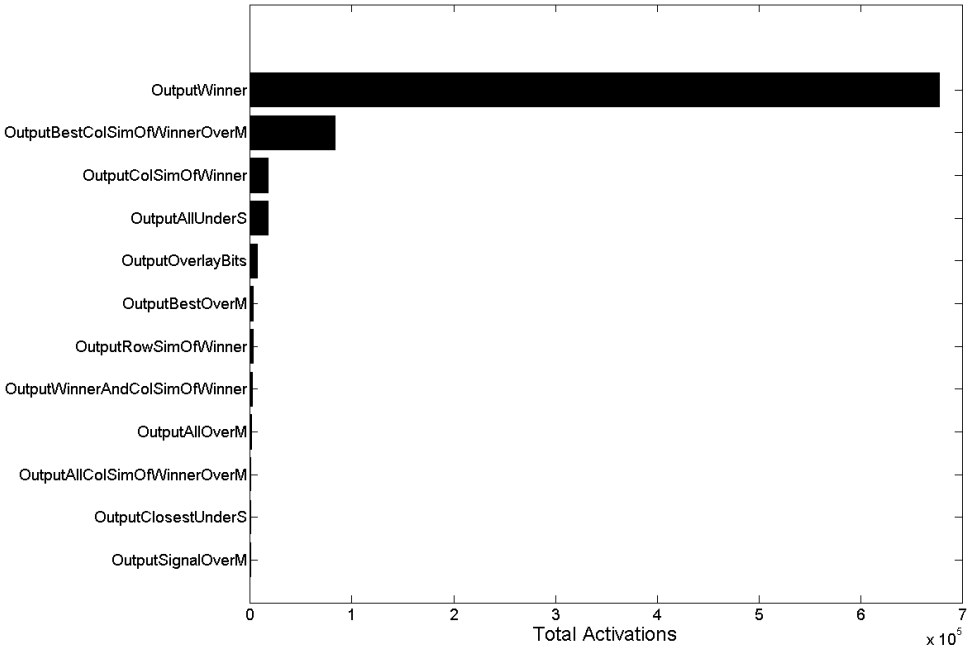


Figure 5.32 Sum of output script activations for the left-or-right problem in a noisy simulation.

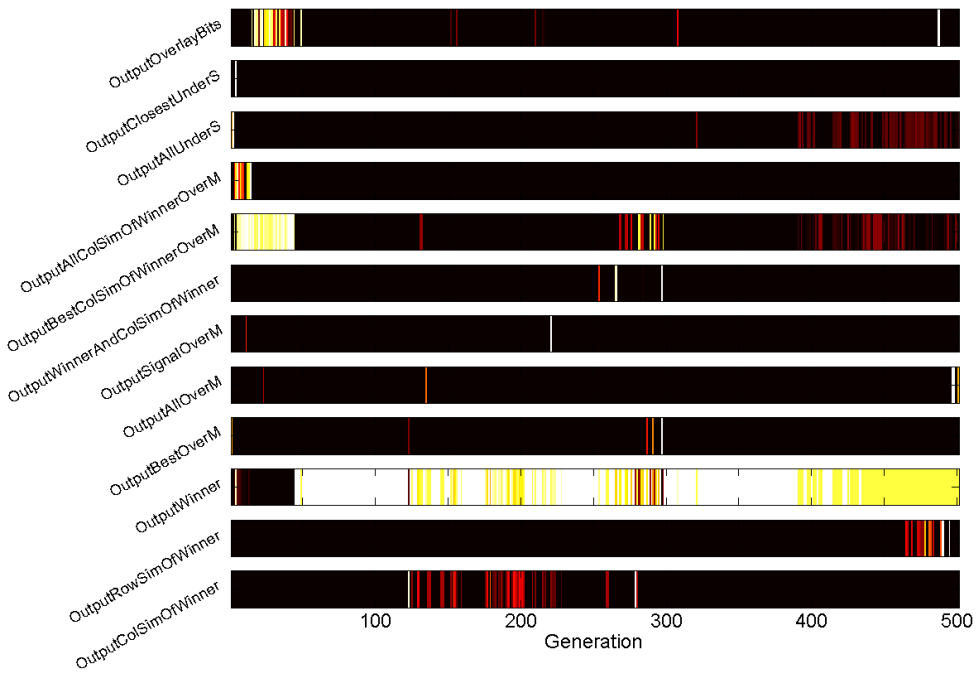


Figure 5.33 Output script activations in the best solution in each generation for the left-or-right problem in a noisy simulation.

match is still the correct output; secondly, the noise may increase the matches to other SOM nodes such that their match is over the threshold M, causing the classifier node to output multiple categories.

5.3.1 Comparison of Self-Organising Map Implementations

One of the advantages of using the SOP implementation, as described in Equation 2.10 to determining the SOM is its robustness to noise, compared to using a separation implementation as described in Equation 5.2. Consider the scenario illustrated in Figure 5.34, where an 8-bit input pattern has had a single noise bit added.

Using the SOP implementation the match is still 100% with the added noise. Using the separation implementation the match falls to 66%.

5.4 SOLUTION ROBUSTNESS

An investigation into the robustness of solutions and improving robustness was undertaken. Robustness is defined as the ability of a particular network to solve a problem repeatedly, given different training sequences. This is an important attribute given the highly stochastic nature of the training input sequences. Some solutions may be more sensitive to training sequence variations than others. By the nature of a GA,

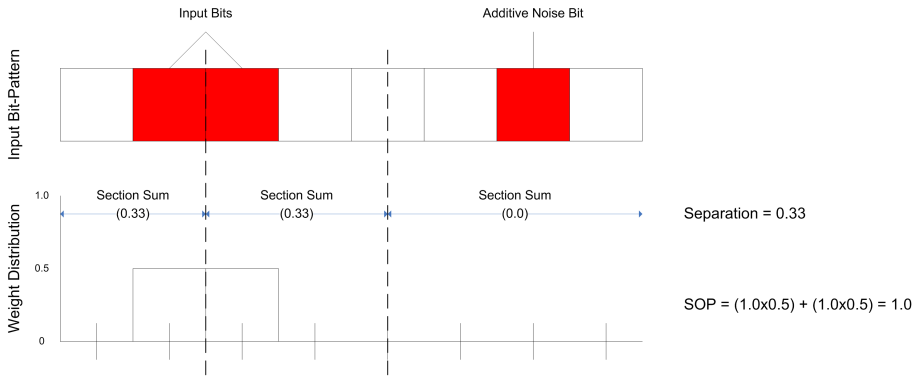


Figure 5.34 Comparison of SOP and separation SOM implementations for noise immunity.

over time solutions that consistently perform well should be selected for. However, just because the maximum fitness reached during a simulation was 100% it was unclear how robust that solution was. In a GA population there is likely to be many highly similar individuals, particularly after the GA has been running for a long time and is converging on an optimum. Consider a Dura population where the five best individuals have essentially the same script tables. If the individuals can achieve a fitness of 100%, 40% of the time, through elitism, crossover between the similar individuals, and minor mutations, the five similar individuals can persist in the population without improving. To encourage robustness of solutions, each individual was simulated over n trials and the fitness of each individual was the average fitness.

Table 5.2 provides the fitness statistics for three different solutions to the orbiting point problem, from three different simulations; Figure 5.35 provides box and whisker plots for the fitness statistics. The simulations were conducted such that each individual was tested over either 1, 2, or 3 trials. An individual's fitness was then determined by an average of its fitness from each trial. Each simulation was run for 500 generations.

Table 5.2 Data statistics of fitness for different solutions to the orbiting point problem. Each solution was found by simulating for 500 generations using 1, 2, or 3 trials per generation.

Num. Trials	**1**	**2**	**3**
Max	1	1	1
Min	0.3500	0.6500	0.7630
Mean	0.8469	0.8969	0.9661
Median	0.9187	0.9500	1
Range	0.6500	0.3500	0.2370
σ	0.1898	0.1184	0.0616

It can be seen that each of the solutions could potentially achieve a fitness of 100%.

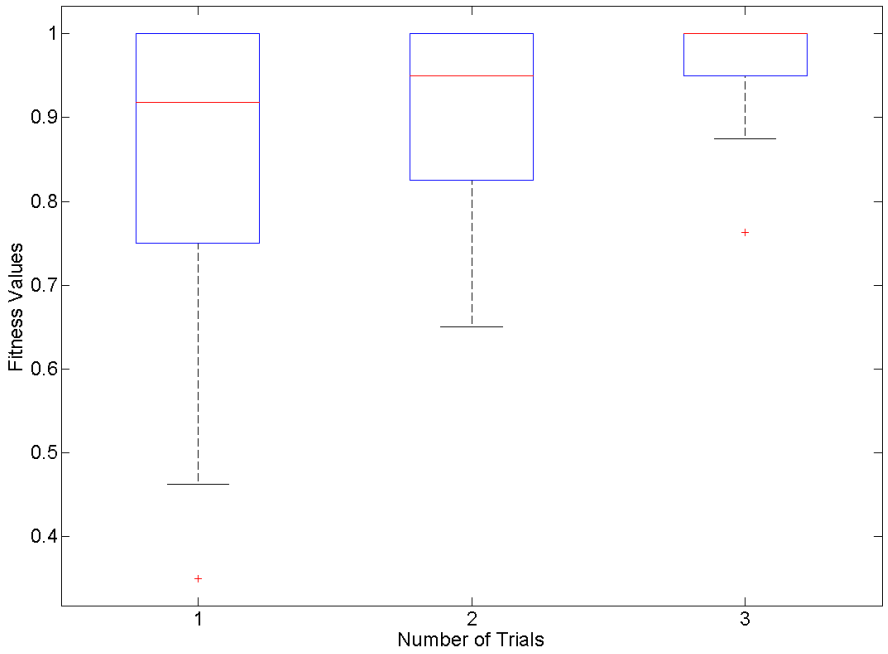


Figure 5.35 Fitness box and whisker plots for three different solutions to the orbiting point problem. Each solution was found by simulating for 500 generations using 1, 2, or 3 trials per generation.

From Table 5.2 and Figure 5.35 it can be seen that there was a positive relationship between robustness and number of trials. As the number of trials increased, the minimum, median, and lower quartile fitness increased, while the standard deviation and range decreased. This result showed that selection for robustness can be introduced into the GA.

The reasons for improved robustness were subtle differences in the selected scripts, and also fine tuning of the scripts' parameters. One of the distinctions between the different solutions was the emergence of multiple scripts during training. `OutputWinner` and `OutputAllOverM` was used while training the two most robust solutions. `OutputBestOverM` was the only output script used in training the least robust solution. One of the most important contributors to robustness of solutions was the SOM convergence rate. If the convergence rate is too fast then a short sequence of "bad" input samples will cause the map to be a poor representation of the input space. In the two most robust solutions the rate of convergence of the SOMs was more refined. This was achieved in two ways; firstly, by using the script `OutputWinner` rather than `AllOverM` when the SOM matches were still a poor representation of the input space; secondly, probably the main reason for the increased robustness was fine tuning of the script instruction parameters, particularly the SOM learning rates and the threshold M in `OutputAllOverM`.

5.5 GENETIC ALGORITHM MECHANISM TRENDS

The GA had five main mechanisms by which it could produce solutions in each generation: elitism, SSC, mutation, spawning, and crossover. An investigation was conducted to discover which mechanisms were responsible for producing the best and worst solutions, and how this mechanism use changed at different stages of a simulation. In this investigation the best and worst solutions were determined by their ranked fitness. To generate Figure 5.37 and Figure 5.39, a moving average of the proportions of mechanisms used to produce the 10 best solutions at each generation (from a population of 50) was plotted. Similarly, Figure 5.40 was generated by plotting the moving average of mechanism proportions producing the worst individual at each generation. Figure 5.36 illustrates the fitness convergence for a left-or-right problem simulation, Figure 5.37 illustrates the mechanism trends.

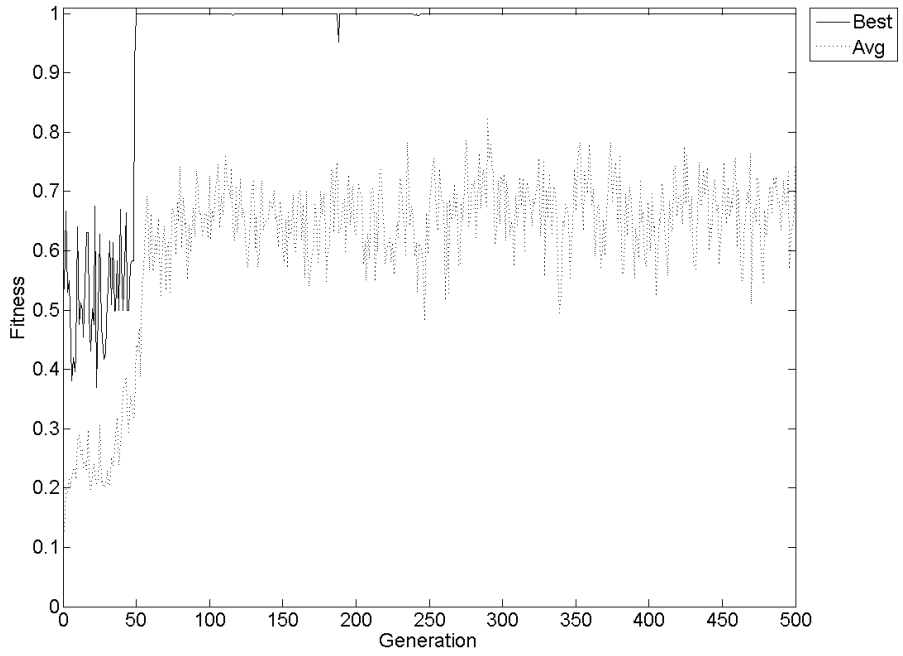


Figure 5.36 Fitness convergence for the left-or-right problem.

Crossover contributed between 10-20% of the fittest individuals after generation 60. This is likely due to lack of variation in the best individuals, resulting in children that are identical or very similar to the existing best solutions.

It is also likely that the mutations producing the fittest solutions after convergence were those which did not affect functional parts of the previous best solutions. The emergence of elitism as the primary mechanism producing the fittest individual can also be seen. Figure 5.38 shows the fitness convergence for a overlapping lines problem

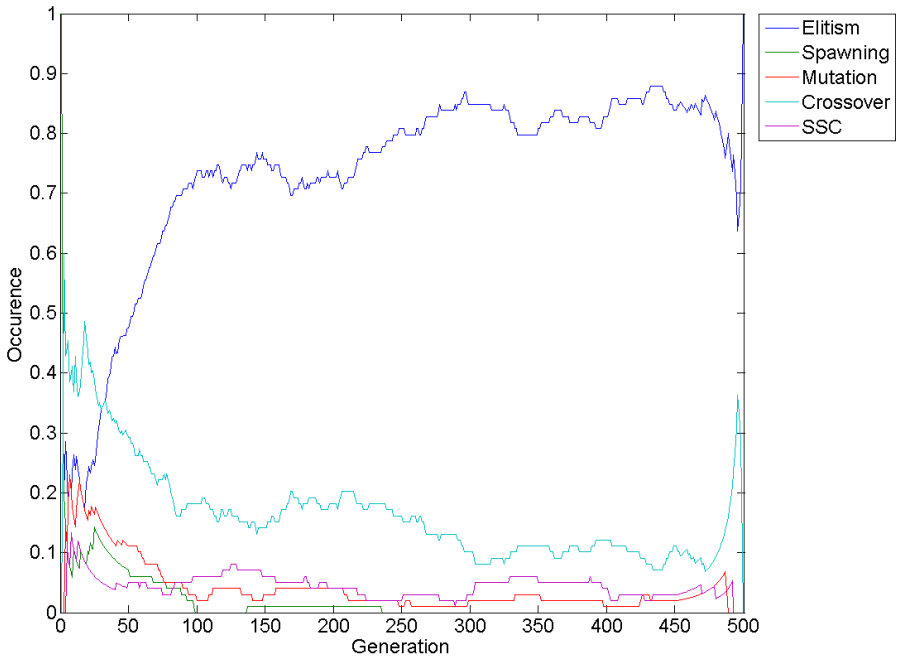


Figure 5.37 Relative prominence of mechanism trends producing the best individuals in each generation for the left-or-right problem.

simulation, Figure 5.39 shows the mechanism trends.

This particular simulation only converged to a fitness of around 60%. Unlike the left-or-right simulation, which converged to 100%, we do not see the emergence of elitism as the dominating mechanism producing the fittest individual. Instead we see crossover representing a fairly similar presence, or slightly greater, to elitism, with mutation also having an increased percentage.

Figure 5.40 illustrates the mechanism trends producing the worst individual for a rotating line simulation which converged to a fitness of 100% after about 10 generations. It can be seen from this graph that spawning is by far the most common contributor. This makes sense because the individuals are spawned at random and there are far more “bad” script combinations than “good” script combinations. It is still important to generate random individuals as they may create better individuals very occasionally.

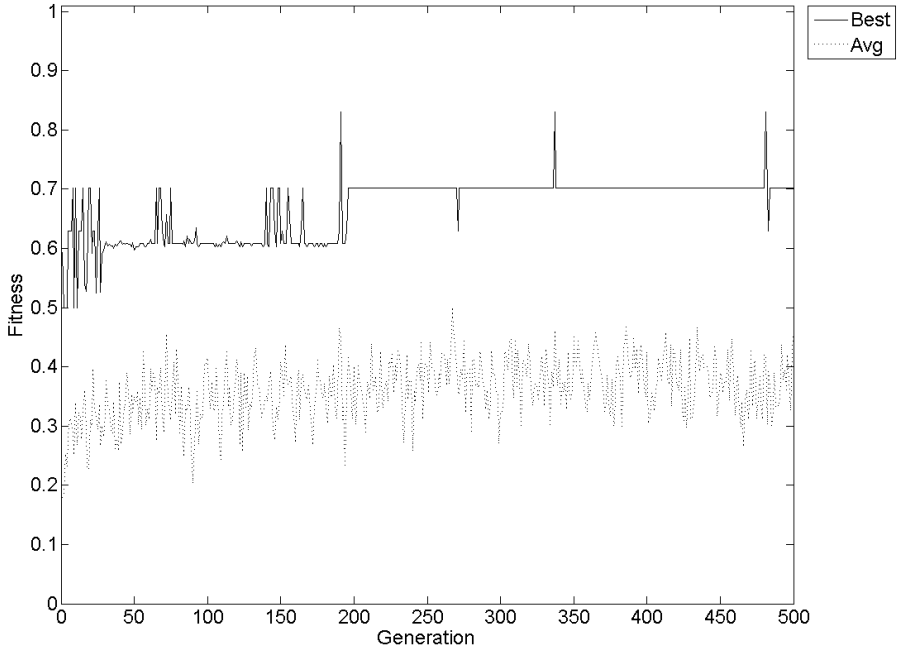


Figure 5.38 Fitness convergence for the dual orbiting points problem.

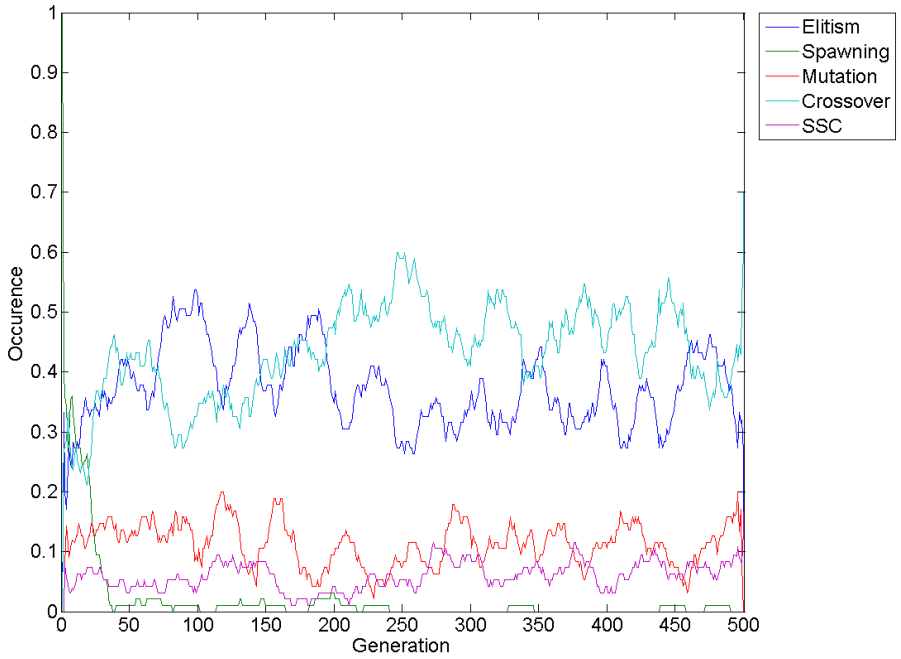


Figure 5.39 Relative prominence of mechanism trends producing the best individuals in each generation for the dual orbiting points problem.

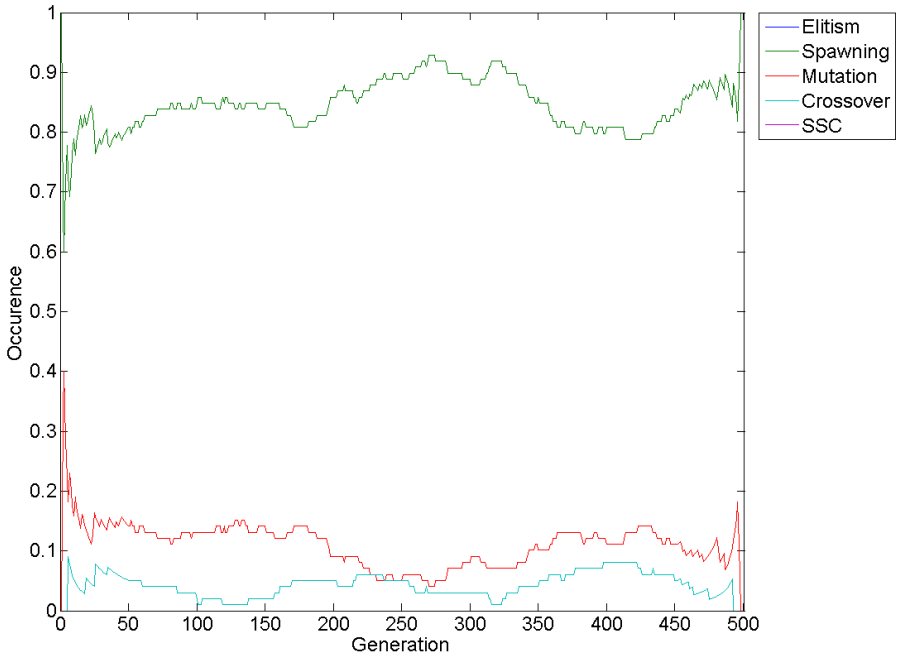


Figure 5.40 Relative prominence of mechanism trends producing the worst individual in each generation for the rotating line problem.

Chapter 6

CONCLUSIONS

This chapter gives suggestion for future improvements to the Dura project and discusses general conclusions. The conclusions drawn here are not just a reiteration of the results which are given and discussed in Chapter 5.

6.1 FUTURE WORK

The main area for future development suggested here is sequential learning. Three possible implementations are discussed in the following: SOM trajectories, echo state networks (ESNs), and output timing. The other area for future improvement suggested here is in greater GA control.

6.1.1 Sequential Learning

One of the main future improvements for the Dura network would be to develop it into a sequential learner which maintains a record of past inputs. It is commonly accepted that sequence learning is an important component of many intelligent systems and that sequence learning is arguably the most prevalent form of human and animal learning [51]. The basic Dura framework outlined in this thesis could be extended so that classifier nodes can use past inputs to classify the current input and also to make predictions about future inputs.

In the static Dura network, information becomes more abstract in dimensionality as it propagates up the hierarchy. A sequential Dura network could also look further ahead in time as information propagates up the hierarchy. The bottom levels would be looking at the current input and the top level could be making predictions about the input several time steps later.

In the real world, the way humans experience it, we rarely see events occur instantaneously then disappear; there is always a cause and effect. For example: if I drop a glass, I can predict it is going to fall to the floor and may shatter. Sequential information is extremely important when classifying/predicting certain real world data. For example: predicting/classifying abnormalities in electro-cardiogram signals.

Several ways of exploiting temporal information in the Dura network are considered, including: SOM trajectories, ESNs, and output timing.

6.1.1.1 Output Timing

Many studies consider pulse-coupled networks with spike-timing as an essential component in information processing by the brain [52] and much research has been done in this area. New scripts could be implemented to cause classifier nodes to generate time coded *outputs* as well as static *outputs*. Classifier nodes would also need to maintain a short term memory to decode inputs. In this way classifier nodes would have a far greater vocabulary for communication between levels and may exhibit spiking behaviours similar to documented biological spiking neurons as shown in [53].

6.1.1.2 Self-Organising Map Trajectories

Another way to incorporate temporal information is by looking at trajectories formed in SOMs as used in [3]. A SOM trajectory is formed from a memory of which SOM nodes have been activated in the past and the path that was taken between the nodes. The trajectories are mapped and can be compared to previous trajectories to classify the current input and predict future inputs.

6.1.1.3 Echo State Networks

One of the results that became clear during the simulation and experimentation was that the SOM implementation was a limiting factor in the type of problem that the Dura network could solve. One interesting development in ANN research is ESNs [54, 55], similar to liquid state Machines (LSMs) [56]. ESNs have been shown to predict highly non-linear relationships using only a small amount of training [57]. ESNs became an avenue of serious investigation for this project, and therefore some implementation detail is provided in the following. An ESNs output depends on current and previous inputs and are usually used for prediction of time-series problems, rather than classification. In real-world terms, it is hard to distinguish between classification and prediction. If a person can identify an object, it stands to reason that it is still going to be there a fraction of a second later and that it is still going to be the same object. This idea of a sequential learning classification network led to a whole new direction of research which went beyond the original scope of this project. However, preliminary results were gathered and showed potential usefulness. A brief overview of ESNs is given here followed by some preliminary results.

ESNs have an input weight vector, output weight vector, and an internal node reservoir matrix. The internal reservoir matrix defines the connection strengths between each of the internal nodes, the connections in this matrix are usually sparse.

The sparseness of this matrix allows different regions of excitation to emerge in the reservoir. Each input node, $\mu(n)$, is mapped through a weight vector to each internal node, $\mathbf{x}(n)$, each internal node is mapped through a weight vector, $\mathbf{w}_{out}(n)$ to the output nodes, $\mathbf{y}(n)$. The input is sometimes connected to the output through an additional weighted connection. When the network is created all of the weights are generated at random. At each time step an input is presented and the output produced is compared to a teaching signal. The error between the output and the teaching signal is found and the output weight vector is updated in order to minimise the mean squared error. The network error, the equation to be minimised, is given in Equation 6.1.

$$\epsilon_{train}(n) = (tanh)^{-1} y_{teach}(n) - \mathbf{w}_{out}(\mu_{teach}(n), \mathbf{x}(n)) \tag{6.1}$$

A more detailed explanation of ESN implementation can be found in *Adaptive nonlinear system identification with echo state networks* [57] by Herbert Jaeger. Over time for repeated sequences the network will learn to predict the training signal for uncorrelated inputs. Figure 6.1 illustrates an ESN configuration with a single input and output node.

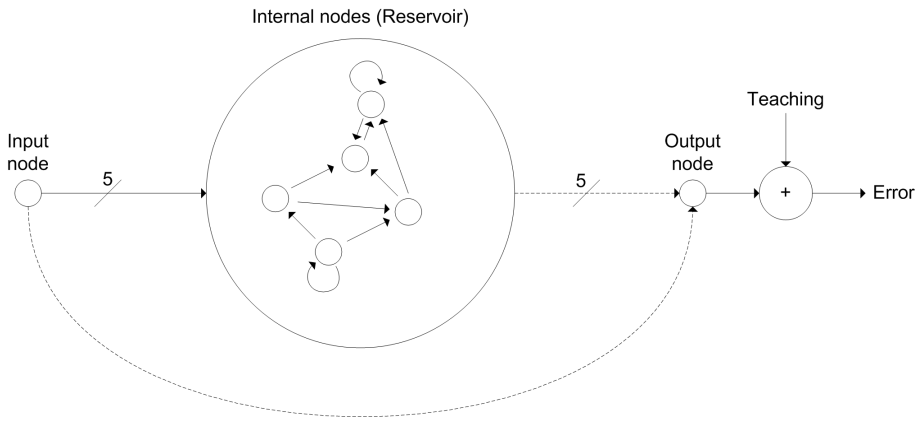


Figure 6.1 Illustration of an ESN configuration. Solid lines indicate fixed connections, dashed lines indicated adaptable connections.

An ESN implementation written in C++ was tested to see how well a 32 input, 8 output ESN would converge to an 8 node training signal using a 50 node reserviour. Figure 6.2 illustrates the absolute error signal between the output nodes and the training nodes.

Figure 6.2 shows that the error reduces to around 10% after ten presentations. An ESN implementation was then imported into the Dura project. All ESN internal values were maintained as float values between -1.0 and 1.0. These float values were mapped to a 0 or 1 bit using some tunable threshold. The ESN implementation was then used

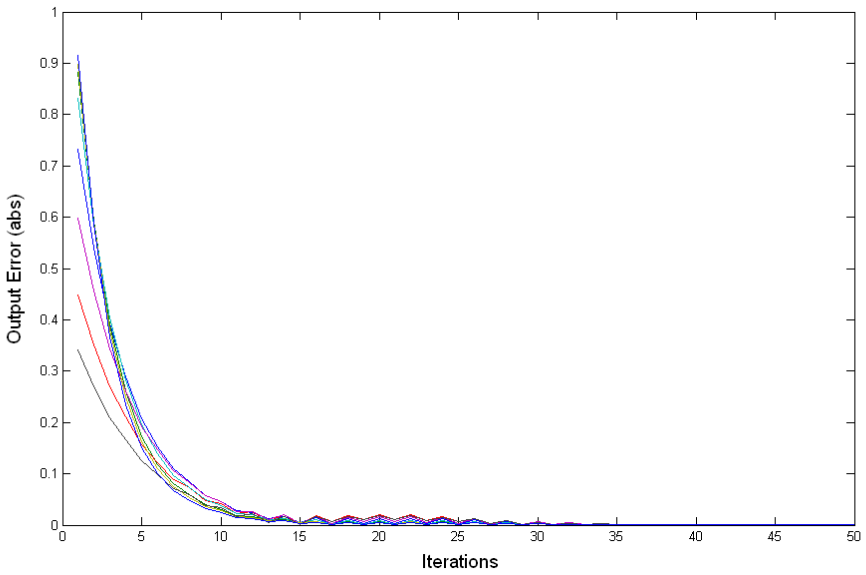


Figure 6.2 Convergence of ESN outputs. Each line on the graph represent the absolute error for a different output node.

to replace the SOM for a single classifier node network presented with the overlapping lines problem. The 32 node ESN input was converted from the *input* to the classifier node. The ESN then generated an 8 node output which was converted and compared to the 8-bit *training* label bit-pattern. Figure 6.3 shows the resulting output label map after training the classifier with 1000 input presentations.

It can be seen from Figure 6.3 that the ESN implementation has adjusted to output the important relationship between inputs which is the diagonal line where the input lines overlap. This result suggested that ESNs could be a useful tool for future development.

The sort of problem that could be investigated by a sequential learner could be a problem space similar to the overlapping lines where the goal is to count the number of lines on the retina. Consider the overlapping lines problem constrained so that each line could only move 1 pixel left or right in each time step and each line could exit the retina entirely at either side and only re-enter from either side. A static classification network would not be able to tell the difference between two lines overlapping and a single line on the retina. If outputs are effected not only by the current input but by previous ones as well, a sequential implementation such as, ESNs, should be able to make this distinction. One of the important results of this investigation was that the classifier node structure itself, was well factored enough to allow implementation of a new internal mechanism.

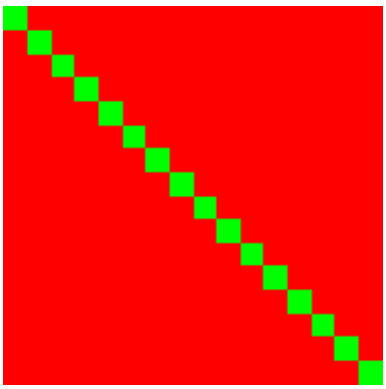


Figure 6.3 Overlapping lines output label map with an ESN implementation.

6.1.2 Greater Genetic Algorithm Control

Another future improvement for this project would be to give even greater control to the GA. The GA could be used to optimise many facets of the Dura network: network configuration, number of training samples, number of SOM nodes, number of testing samples, the size of the *inputs* and *outputs*, and number of trials to determine fitness. During the course of simulations there were still some parameters, under manual control, whose complex interactions could have impacted on the simulation results. Therefore, there could be benefits to removing as much manual control as possible. Some of the simulation parameters could be controlled by the GA, but others such as the GA parameters themselves (for example, `populationSize`) must be chosen manually.

There may however be potential difficulties in giving the GA greater control. Firstly, due to the implementation in C++ of features such as network configuration, the parameters must be specified at compile time. Therefore, to pass data from the GA would require some restructuring of code. Much care was taken to make the code generally expandable and maintainable, however sometimes restructuring of code will be necessary. Secondly, increasing the GAs search space will inevitably increase the simulation execution time, increase the average time required to converge on a good solution, and also increase the likelihood of sub-optimal solutions. While there are definitely potential benefits to automating more and more of the Dura components, it is beyond the initial project scope, which was to investigate the scripted instruction space optimized with a GA.

6.2 GENERAL CONCLUSIONS

This project has demonstrated that a hierarchical homogeneous scripted vector classification network can successfully solve some visual classification problems. A discussion

of which scripts were used and how the problems were solved is given in Section 5.2 and are not reiterated here. This research showed that a GA is a powerful tool for optimizing systems that are highly non-linear and consist of parameter relationships that are difficult to predict. It was also seen that the GA could be applied to specifically improve noise immunity and robustness of solutions. One of the best features and successes of this investigation is the generality of the Dura classification system. Any problem space that can be conceived and translated into the code can then be tested without any changes to the program itself.

During the course of this project a common theme was that the limiting factor in determining the networks success was the SOM. The SOP SOM implementation could only discern relatively simple relationships in simple problem spaces. This was improved by using the separation mechanism to determine the winner. The type of problems that could be solved depended largely on the implementation of the SOM matching mechanism. A more complex input parser that could derive non-linear relationships between the input and the output could be beneficial, either replacing the SOM or perhaps as a second stage. This is why an ESN implementation is thought to be a promising avenue of investigation.

The Dura network was intended to handle sparse input problem spaces better than dense input spaces. This was due to the fact that a classical SOM only outputs 1 node for each input. With a data compression of $\frac{1}{8}$ in the SOM, input space densities around $\frac{1}{8}$ were predicted to be most appropriate and successful. In practice this didn't eventuate, because of the ability of the SOMs to output all nodes over a certain match. Therefore, the middle and top level retinas were often quite dense. This was essentially a result of the SOM winner found by separation implementation, and was in fact crucial to solving the overlapping lines problem, *cf.* Section 5.2.4.

During the course of this research the conclusion was reached that static input classification is not sufficient to model real world learning and that sequential learning would be an interesting extension. It is likely that there is a need for some kind of sequential component and that there is plenty of potential to add sequential learning to the Dura project, as discussed in Section 6.1.

One of the issues with using a GA for this network is the structure of the `ScriptTokens`. For example: if the `code` is changed then the rest of the tunable parameters may become meaningless. Similarly when crossover occurs between two script tables with different `codes` at the same index then the parameters have different purposes. The results did show however that the GA could in fact consistently converge on good solutions, demonstrating the versatility of a GA with carefully chosen parameters.

One of the successes of this project was in implementing and maintaining a complex C++ program. The code produced was highly object oriented and well factored. This was demonstrated by the way that the internal mechanisms of the classifiers could be

converted to use ESNs as a primary internal mechanism. The program involved many thousands of lines of code and experience gained during this project will be invaluable in projects to come.

Human and animal learning is an extremely complex subject and very little is still known. For example: despite 100 years of study the transformations dendrites perform on their inputs remain poorly understood [58]. A general observation about investigating complex concepts, such as learning, is that there are two very different approaches to take. One approach is to attempt derive mathematically tractable models from well understood methods in order to attempt to solve the problem. The other approach is to create an arbitrarily complex system using highly non-linear methods and then to empirically test its ability to solve the problem. There are pros and cons for each of these approaches. In the first approach when a solution is found it may be easily proved and formalised, however in the current field of neuro-science very little is known about real learning systems. There is large evidence to support that their interactions are incredibly complex and difficult to formalize. Therefore this approach may never succeed beyond trivial learning examples. The problem with the second approach however, is that even if the system does replicate some desired learning behaviour, it may still be extremely difficult to discover the important mechanisms which are producing those effects.

Another important conclusion which I think cannot be completely ignored, is that for increasingly complicated and innovative systems that are not well understood, there also increases the potential for human error. A system can be devised where it is uncertain what the output *should* be, therefore when the results are undesirable it is unclear whether this is a result of the conceptual system or the *human implementation* of the system.

This was a fairly open-ended and extremely interesting project. During the course of this research many new ideas have emerged and implemented with different degrees of success. The main objectives for this project have been met and there now exists a framework for a classifier network that can adapt to different problem spaces which still has the potential for further development.

REFERENCES

[1] G. Foresti, V. Murino, C. Regazzoni, and A. Trucco. A voting-based approach for fast object recognition in underwater acoustic images. *IEEE J. Ocean. Eng.*, 22(1), 1997.

[2] Y. Shimshoni and N. Intrator. Classification of seismic signals by integrating ensembles of neural networks. In *Proceedings of the International Conference on Neural Information Processing, ICONIP*, Hong Kong, China, 1996.

[3] J. Kangas, K. Torkkola, and M. Kokkonen. Using SOMs as feature extractors for speech recognition. In *Proceedings of the International Conference on Acoustics, Speech and Signal Processing, ICASSP*, Piscataway, NJ, USA, 1992. IEEE Service Center.

[4] P. Tan, M. Steinbach, and V. Kumar. *Introduction to Data Mining.* Addison-Wesley, 2006. Ch. 4, pp. 148.

[5] Y. Davidor. A genetic algorithm applied to robot path-planning. Technical Report CS90-01, The Weizmann Institute of Science - Department of Applied Mathematics and Computer Science, Rehovot, Israel, 1990.

[6] R. Bauer. *Genetic Algorithms and Investment Strategies.* New York: John Wiley & Sons Inc., 1994.

[7] W. G. Lin and S. S. Wang. A new neural model for invariant pattern recognition. *Neural Networks*, 9:899–913, October 1996.

[8] C. K. Oh and G. J. Barlow. Autonomous controller design for unmanned aerial vehicles using multi-objective genetic programming. In *Proceedings of the 2004 Congress on Evolutionary Computation, CEC*, pages 1538–1545, Portland, OR, USA, June 2004.

[9] K. F. Man, K. S. Tang, S. Kwong, and W. A. Halang. *Genetic Algorithms for Control and Signal Processing.* London: Springer-Verlag, 1997.

[10] C. Muller et al. A neural network tool for forecasting french electricity consumption. In *Proceedings of the World Congress on Neural Networks, WCNN*, volume 1, pages 360–365, 1994.

[11] H. A. Rowley, S. Baluja, and T. Kanade. Neural network-based face detection. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 20(1):23–38, 1998.

[12] I. Yoo. Visualizing windows executable viruses using self-organizing maps. In *Proceedings of the 2004 ACM workshop on Visualization and Data Mining for computer Security, VizSEC/DMSEC*, pages 82–89, Washington, DC, USA, 2004. ACM Press.

[13] J. Lampinen and O. Taipale. Optimization and simulation of quality properties in paper machine with neural networks. In *Proceedings of the International Conference on Neural Networks, ICNN*, volume 1, pages 3812–3815, 1994.

[14] P. Cosi, G. De Poli, and G. Lauzzana. Timbre classification by nn and auditory modeling. In *Proceedings of the International Conference on Artificial Neural Networks, ICANN*, volume 2, pages 925–928. London: Springer, 1994.

[15] S. V. Kartalopoulos. *Understanding Neural Networks and Fuzzy Logic: Basic Concepts and Applications*. Wiley-IEEE Press, 1997. pp. 168.

[16] E. Alpaydin. *Introduction to Machine Learning (Adaptive Computation and Machine Learning)*. MIT Press, 2004.

[17] C. M. Bishop. *Pattern Recognition and Machine Learning*. Springer, 2007.

[18] T. M. Mitchell R. S. Michalski, J. G. Carbonell. *Machine Learning: An Artificial Intelligence Approach, Volume I*. Tioga Publishing Company, 1983.

[19] T. M. Mitchell R. S. Michalski, J. G. Carbonell. *Machine Learning: An Artificial Intelligence Approach, Volume II*. Morgan Kaufmann, 1986.

[20] R. S. Michalski Y. Kodratoff. *Machine Learning: An Artificial Intelligence Approach, Volume III*. Morgan Kaufmann, 1990.

[21] T. M. Mitchell R. S. Michalski, J. G. Carbonell. *Machine Learning: A Multistrategy Approach, Volume IV*. Morgan Kaufmann, 1994.

[22] A. J. Owens L. J. Fogel and M. J. Walsh. *Artificial Intelligence through Simulated Evolution*. New York: John Wiley, 1966.

[23] I. Rechenberg. *Evolutionsstrategie: Optimierung technischer Systeme nach Prinzipien der biologischen Evolution*. Stuttgart: Fromman-Holzboog, 1973.

[24] J. H. Holland. *Adaptation in Natural and Artificial Systems: An Introductory Analysis with Applications to Biology, Control, and Artificial Intelligence.* MI: University of Michigan Press, 1975.

[25] J.R. Koza. *Genetic Programming: On the Programming of Computers by Means of Natural Selection.* MIT Press, 1992.

[26] C. Darwin. *On the Origin of Species.* London: John Murray, 1859.

[27] S. G. Ficici, O. Melnik, and J. B. Pollack. A game-theoretic investigation of selection methods used in evolutionary algorithms. In *Proceedings of the 2000 Congress on Evolutionary Computation, CEC*, page 880, La Jolla, CA, USA, 2000. IEEE Press.

[28] M. Mitchell. *An Introduction to Genetic Algorithms.* MIT Press, 1996.

[29] T. Blickle and L. Thiele. A comparison of selection schemes used in genetic algorithms. Technical Report 11, Swiss Federal Institute of Technology (ETH), Zurich, Switzerland, 1995.

[30] A. Sokolov and D. Whitley. Unbiased tournament selection. In *Proceedings of the Conference on Genetic and Evolutionary Computation, GECCO*, pages 1131–1138, Washington, DC, USA, 2005. ACM Press.

[31] J. Gottlieb and T. Kruse. Selection in evolutionary algorithms for the traveling salesman problem. In *Proceedings of the ACM Symposium on Applied Computing, SAC*, pages 415–421, Como, Italy, 2000. ACM Press.

[32] J. Sarma and K. De Jong. An analysis of the effects of neighborhood size and shape on local selection algorithms. In *Parallel Problem Solving from Nature IV, PPSN*, pages 236–244, Berlin, Germany, 1996. Springer.

[33] T. Blickle and L. Thiele pp. 48. A comparison of selection schemes used in genetic algorithms. Technical Report 11, Swiss Federal Institute of Technology (ETH), Zurich, Switzerland, 1995.

[34] S. Legg and M. Hutter. Fitness uniform deletion: a simple way to preserve diversity. In *Proceedings of the 2005 Conference on Genetic and Evolutionary Computation, GECCO*, pages 1271–1278, Washington, DC, USA, 2005. ACM Press.

[35] W. M. Spears. *Evolutionary Algorithms: the role of mutation and recombination.* Springer, 2000.

[36] W. M. Spears. Crossover or mutation? In *Foundations of Genetic Algorithms 2*, pages 221–233, San Mateo, CA, USA, 1993.

[37] F. J. Ovalle-Mart, J. S. Gonz, and I. Stojmenovi. A parallel hill climbing algorithm for pushing dependent data in clients-providers-servers systems. *Mobile Network Applications*, 9(4):257–264, 2004.

[38] S. Nahar, S. Sahni, and E. Shragowitz. Simulated annealing and combinatorial optimization. In *Proceedings of the 23rd ACM/IEEE Conference on Design Automation, DAC*, pages 293–299, Las Vegas, NV, USA, 1986. IEEE Press.

[39] K. E. Parsopoulos and M. N. Vrahatis. Particle swarm optimization method in multiobjective problems. In *Proceedings of the ACM Symposium on Applied Computing, SAC*, pages 603–607, Madrid, Spain, 2002. ACM Press.

[40] A. H. Wright and J. N. Richter. Strong recombination, weak selection, and mutation. In *Proceedings of the Conference on Genetic and Evolutionary Computation, GECCO*, pages 1369–1376, Seattle, WA, USA, 2006. ACM Press.

[41] T. Kohonen. *Self-Organizing Maps*. Springer series in information sciences, 30. Berlin ; New York : Springer, 2nd ed. edition, c1997.

[42] A. K. Jain, M. N. Murty, and P. J. Flynn. Data clustering: a review. *ACM Comput. Surv.*, 31(3):264–323, 1999.

[43] T. Kohonen, E. Oja, O.Simula, A. Visa, and J. Kangas. Engineering applications of the self-organizing map. In *Proceedings of the IEEE*, volume 84, pages 1358–1384, October 1996.

[44] T. Kohonen. *Self-Organizing Maps - pp. 88*. Springer series in information sciences, 30. Berlin ; New York : Springer, 2nd ed. edition, c1997.

[45] M. Jordan and R. Jacobs. Modular and hierarchical learning systems. In *The Handbook of Brain Theory and Neural Networks*, Cambridge, MA, USA, 1995. MIT Press.

[46] S. R. Waterhouse and A. J. Robinson. Classification using hierarchical mixtures of experts. In *Proceedings of the IEEE Workshop on Neural Networks for Signal Processing IV*, pages 177–186, Long Beach, CA, USA, 1994. IEEE Press.

[47] H. Yang and M. Palaniswami. On the issue of neighborhood in self-organizing maps. In *Proceedings of the ACM/SIGAPP Symposium on Applied Computing, SAC*, pages 412–416, Kansas City, MO, USA, 1992. ACM Press.

[48] B. Spitzak. Fast light toolkit. [Online]. Available: http://www.fltk.org/.

[49] D. L. Willick, D. L. Eager, and R. B. Bunt. Disk cache replacement policies for network fileservers. In *Proceedings of the International Conference on Distributed Computing Systems*, pages 2–11, 1993.

[50] T. Kohonen. Self-organized formation of topologically correct feature maps. In *Neurocomputing: foundations of research*, pages 509–521, Cambridge, MA, USA, 1988. MIT Press.

[51] R. Sun and C. L. Giles. Sequence learning: From recognition and prediction to sequential decision making. *IEEE Intelligent Systems*, July 2001.

[52] E. M. Izhikevich. Which model to use for cortical spiking neurons? *IEEE Transactions on Neural Networks*, 15(5):1063–1070, 2004.

[53] E. M. Izhikevich. Which model to use for cortical spiking neurons? - pp. 1064 - fig. 1. *IEEE Transactions on Neural Networks*, 15(5):1063–1070, 2004.

[54] H. Jaeger. The "echo state" approach to analysing and training recurrent neural networks. Technical Report 148, German National Research Center for Information Technology, 2001.

[55] H. Jaeger and H. Haas. Harnessing Nonlinearity: Predicting Chaotic Systems and Saving Energy in Wireless Communication. *Science*, 304(5667):78–80, 2004.

[56] W. Maass, T. Natschlger, and H. Markram. Real-time computing without stable states: A new framework for neural computation based on perturbations. *Neural Computation*, 14(11):2531–2560, 2002.

[57] H. Jaeger. Adaptive nonlinear system identification with echo state networks. In *Advances in Neural Information Processing Systems*, pages 593–600, Cambridge, MA, USA, 2003. MIT Press.

[58] M. Häusser and B. Mel. Dendrites: bug or feature? *Current Opinion in Neurobiology*, 13:372–383, 2003.